



# DesignWare Cores MIPI I3C Slave-Lite Controller

## Databook

---

*DWC MIPI I3C CONTROLLER (SLAVE-LITE CONFIGURATION) - Product Code: B611-0*

Synopsys confidential document, provided to CodaDevice  
Semiconductor under NDA, do not redistribute

## Copyright Notice and Proprietary Information

© 2020 Synopsys, Inc. All rights reserved. This Synopsys software and all associated documentation are proprietary to Synopsys, Inc. and may only be used pursuant to the terms and conditions of a written license agreement with Synopsys, Inc. All other use, reproduction, modification, or distribution of the Synopsys software or the associated documentation is strictly prohibited.

### Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

### Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

### Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at <https://www.synopsys.com/company/legal/trademarks-brands.html>

All other product or company names may be trademarks of their respective owners.

### Free and Open-Source Software Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

### Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.

[www.synopsys.com](http://www.synopsys.com)

# Contents

Revision History .....	7
Preface .....	9
Databook Organization .....	10
Reference Documentation .....	11
Web Resources .....	12
Customer Support .....	13
<b>Chapter 1</b>	
<b>Product Overview .....</b>	<b>15</b>
1.1 General Product Description .....	16
1.1.1 System Block Diagram .....	16
1.1.2 Applications .....	17
1.1.3 Standards Compliance .....	17
1.2 Slave-Lite Features .....	19
1.2.1 General Features .....	19
1.3 Deliverables .....	22
1.4 Unsupported Features .....	23
<b>Chapter 2</b>	
<b>Architecture .....</b>	<b>25</b>
2.1 Block Diagram of Slave-Lite .....	26
2.2 Description of Slave-Lite Internal Blocks .....	27
2.2.1 Bus Monitor .....	27
2.2.2 Receive Control .....	27
2.2.3 Transmit Control .....	27
2.3 Description of Slave-Lite Bus Interfaces .....	28
2.4 Functionality of Slave-Lite .....	29
2.4.1 Overview of Slave-Lite Functionality .....	29
2.4.2 Description of Slave-Lite Functionality .....	29
2.5 Sub-Address Generation .....	30
2.5.1 Overview of Sub-Address Generation .....	30
2.5.2 Description of Sub-Address Generation .....	30
2.6 Hot-Join Request Generation .....	32
2.6.1 Overview of Hot-Join Request Generation .....	32
2.6.2 Asserting <code>hj_req_now</code> .....	32
2.6.3 De-asserting <code>hj_req_now</code> .....	32

2.7	Slave Interrupt Request Generation	35
2.7.1	Overview of Slave Interrupt Request Generation	35
2.7.2	Asserting <code>sir_req_now</code>	35
2.7.3	De-asserting <code>sir_req_now</code>	35
2.7.4	Asserting <code>sir_req_on_start</code>	36
2.7.5	De-asserting <code>sir_req_on_start</code>	36
2.7.6	Slave Interrupt Request Generation with Data Payload	37
2.7.7	Slave Interrupt Request Generation Flow	40
2.7.8	SIR Generation when Bus is not Idle	42
2.7.9	SIR Generation when Bus is Idle	42
2.8	Transfers	44
2.8.1	Introduction to Private Data Transfers	44
2.8.2	Private Write Transfers with Eight Bit Interface	48
2.8.3	Private Write Transfers with Sixteen Bit Interface	50
2.8.4	Private Read Transfer with Eight Bit Interface	52
2.8.5	Private Read Transfers with 32 Bit Interface	55
2.9	Support for PEC in Private Transfers for JESD403-1 Compliance	58
2.9.1	PEC Generator Algorithm	58
2.9.2	Private Read/Write with PEC Enabled	58
2.9.3	PEC calculation and Check in Slave Mode	59
2.9.4	Timing Diagram of Private Write with PEC Error	60
2.10	Error Handling	61
2.10.1	Connections of Inputs Required for Error Handling	61
2.10.2	Disabling of S0 Error Detection for JESD403-1 Compliance	62
2.10.3	S1 Error Detection and Recovery	63
2.10.4	Parity Error Recovery	63
2.10.5	S2 Error Recovery	65
2.10.6	Asserting <code>slv_par_err_dis</code> and <code>slv_pec_enable</code> for JESD403-1 Compliance	66
2.11	I3C vs I2C Role Selection	67
2.11.1	Conditions for Mode Change	67
2.11.2	Conditions for Glitch Filter Enable/Disable	67
2.11.3	Conditions for Wakeup	67
2.12	CCC Transfers with DWC MIPI I3C Slave-Lite Controller	69
2.12.1	Overview of CCC Transfers with DWC MIPI I3C Slave-Lite Controller	69
2.12.2	Description of CCC Transfers with Slave-Lite	69
2.12.3	CCCs Handled through Internal Registers	70
2.12.4	Handling of ENTDA, GETPID and GETDCR	71
2.13	Support for Vendor Specific CCC with Defining Bytes	72
2.13.1	Timing Diagram of Directed Write CCC without Defining Byte	73
2.13.2	Timing Diagram of Directed Write CCC with Defining Byte	74
2.13.3	Timing Diagram of Broadcast Write CCC without Defining Byte	75
2.13.4	Timing Diagram of Broadcast Write CCC with Defining Byte	76
2.13.5	Timing Diagram of Directed Read CCC	77
2.14	Support for CCCs for JESD403-1 Compliance	79
2.14.1	Handling of SETAASA CCC	79
2.14.2	Handling of DEVCTRL CCC	79
2.14.3	Handling of SETHID CCC	80
2.14.4	Handling of ENEC, DISEC, and RSTDAA CCC	80
2.14.5	Handling of GETSTATUS CCC	80

2.14.6 Handling of DEVCAP CCC .....	81
2.15 Handling of Unsupported CCC .....	82
2.16 PEC Support for CCC for JESD403-1 Compliance .....	83
2.16.1 Broadcast CCC write with PEC Enabled .....	83
2.16.2 Directed CCC write with PEC Enabled .....	83
2.16.3 Timing Diagram of Broadcast Write CCC with PEC Error .....	84
2.16.4 Timing Diagram of Directed Write CCC with PEC Error .....	84
2.17 Bus Reset .....	86
2.17.1 Slave-Lite State after Bus Reset .....	87
<b>Chapter 3</b>	
<b>Parameter Descriptions .....</b>	<b>89</b>
3.1 Basic Configuration Parameters .....	90
3.2 Slave-lite Configuration Parameters .....	91
3.3 Slave Configuration Parameters .....	92
<b>Chapter 4</b>	
<b>Signal Descriptions .....</b>	<b>97</b>
4.1 Debug Interface Signals .....	99
4.2 I3C Interface Signals .....	100
4.3 Slave Interface Signals .....	103
4.4 Scan Interface Signals .....	117
<b>Appendix A</b>	
<b>Area .....</b>	<b>119</b>
A.1 Area .....	120
<b>Appendix B</b>	
<b>Synchronizer Methods .....</b>	<b>121</b>
<b>Appendix C</b>	
<b>CCC Types which Use CCC Channel .....</b>	<b>123</b>

*Synopsys confidential document, provided to GigaDevice  
Semiconductor under NDA, do not redistribute*

# Revision History

The following table provides a summary of changes made to this Databook.

**Table 1**

Date	Release	Description
January 2020	1.00a-lca03	<p>Updated:</p> <ul style="list-style-type: none"> <li>■ “Standards Compliance” on page 17</li> <li>■ “General Features” on page 19</li> <li>■ “Unsupported Features” on page 23</li> <li>■ “Block Diagram of Slave-Lite” on page 26</li> <li>■ “Description of Slave-Lite Bus Interfaces” on page 28</li> <li>■ “IBI with Data Support” on page 42</li> <li>■ “Support for PEC in Private Transfers for JESD403-1 Compliance” on page 58</li> <li>■ “Error Handling” on page 61</li> <li>■ “I3C vs I2C Role Selection” on page 67</li> <li>■ “Overview of CCC Transfers with DWC MIPI I3C Slave-Lite Controller” on page 69</li> <li>■ “Parameter Descriptions” on page 89</li> <li>■ “Signal Descriptions” on page 97</li> </ul> <p>Added:</p> <ul style="list-style-type: none"> <li>■ “JESD403-1 Compliance in DWC_mipi_i3c in Slave-Lite Mode” on page 19</li> <li>■ “Conditions for Mode Change” on page 67</li> <li>■ “Conditions for Glitch Filter Enable/Disable” on page 67</li> <li>■ “Conditions for Wakeup” on page 67</li> <li>■ “Handling of ENTDA, GETPID and GETDCR” on page 71</li> <li>■ 2.13 to 2.17</li> </ul>
March 2019	1.00a-lca02	This is the first LCA version of the Slave Lite shipped with DWC MIPI I3C Master and Slave Controller.

Synopsys confidential document, provided to GigaDevice  
Semiconductor under NDA, do not redistribute



## Preface

---

This document describes the Synopsys' DesignWare Cores MIPI I3C Slave-Lite Controller.

Synopsys confidential document, provided to GigaDevice  
Semiconductor under NDA, do not redistribute

## Databook Organization

The chapters of this databook are organized as follows:

- Chapter 1, “[Product Overview](#)” describes the overall product features of Slave-Lite.
- Chapter 2, “[Architecture](#)” describes the Slave-Lite Block Diagram and its components.
- Chapter 3, “[Parameter Descriptions](#)” describes Slave-Lite configuration options and parameters.
- Chapter 4, “[Signal Descriptions](#)” provides the pinout and describes the I/O for Slave-Lite.
- [Appendix A, “Area”](#), “Area” provides an estimation of Slave-Lite area values.
- [Appendix B, “Synchronizer Methods”](#) describes the synchronizer methods (if any) that are used in Slave-Lite to cross clock boundaries.
- [Appendix C, “CCC Types which Use CCC Channel”](#) describes the CCC types which use CCC channel.

Synopsys confidential document, provided to Synopsys DesignWare Semiconductor under NDA, do not redistribute

## Reference Documentation

- MIPI® Alliance Specification I3CSM, Version 1.0

Synopsys confidential document, provided to GigaDevice  
Semiconductor under NDA, do not redistribute

## Web Resources

- DesignWare IP product information: <http://www.designware.com>
- Your custom DesignWare IP page: <http://www.mydesignware.com>
- Documentation through SolvNet: <http://solvnet.synopsys.com> (Synopsys password required)
- Synopsys Common Licensing (SCL): <http://www.synopsys.com/keys>

Synopsys confidential document, provided to GigaDevice  
Semiconductor under NDA, do not redistribute

## Customer Support

To obtain support for your product:

- First, prepare the following debug information, if applicable:
  - For environment setup problems or failures with configuration, simulation, or synthesis that occur within coreConsultant or coreAssembler, use the following menu entry:
    - File > Build Debug Tar-file

Check all the boxes in the dialog box that apply to your issue. This menu entry gathers all the Synopsys product data needed to begin debugging an issue and writes it to the file <core tool startup directory>/debug.tar.gz.
  - For simulation issues outside of coreConsultant or coreAssembler:
    - Create a waveforms file (such as VPD or VCD)
    - Identify the hierarchy path to the DesignWare instance
    - Identify the timestamp of any signals or locations in the waveforms that are not understood.
- Then, contact Support Center, with a description of your question and supplying the above information, using one of the following methods:
  - For fastest response, use the SolvNet Web site. If you fill in your information as explained below, your issue is automatically routed to a support engineer who is experienced with your product. The Sub Product 1 entry is critical for correct routing.

Go to <http://solvnet.synopsys.com/EnterACall> and click on the link to enter a call.

Provide the requested information, including:

- Customer Tracking Number: Enter your project name. Use the same name for cases related to the same project.
- Product: DesignWare Cores
- Sub Product 1: MIPI Controller
- Sub Product 2: DWC MIPI I3C SLAVE LITE
- Product Version: Version 1.00a-lca03
- Problem Type:
- Issue Severity:
- Problem Title: Provide a short summary of the issue or list the error message you have encountered
- Problem Description: For simulation issues, include the timestamp of any signals or locations in waveforms that are not understood

After creating the case, attach any debug files you created in the previous step.

- Or, send an e-mail message to [support\\_center@synopsys.com](mailto:support_center@synopsys.com) (your e-mail will be queued and then, on a first-come, first-served basis, manually routed to the correct support engineer):
  - Include the Product name, Sub Product 1 name, Sub Product 2 name, and Product Version number in your e-mail (as identified above) so it can be routed correctly.
  - For simulation issues, include the timestamp of any signals or locations in waveforms that are not understood
  - Attach any debug files you created in the previous step.

- Or, telephone your local support center:
  - North America:  
Call 1-800-245-8005 from 7 AM to 5:30 PM Pacific time, Monday through Friday.
  - All other countries:  
<https://www.synopsys.com/support/global-support-centers.html>

Synopsys confidential document, provided to GigaDevice  
Semiconductor under NDA, do not redistribute

# 1

## Product Overview

---

This chapter introduces the Synopsys DesignWare Cores MIPI I3C Slave-Lite Controller features and deliverables. It contains the following sections:


- [“General Product Description”](#) on page 16
- [“Slave-Lite Features”](#) on page 19
- [“Deliverables”](#) on page 22

## 1.1 General Product Description

The Improved Inter Integrated Circuit (I3C) is part of a group of communication protocols defined by the MIPI® Alliance. This specification is developed to ease sensor system design architectures in mobile wireless products by providing a fast, low cost, low power, two-wire digital interface for sensors.

The Slave-Lite controller meets the requirement of I3C protocol as specified in the MIPI I3C Specification.

The Slave-Lite product suits processor-less system, where processor intervention is not required for it's programming. It operates with the SCL clock received from the I3C Master in SDR and HDR-DDR mode of operation. In the Ternary mode of operation, a recovered clock (recovered from ternary symbols) is required for the reception, and an HDR transmit clock is required for transmission. Both HDR transmit clock and HDR recovered clock must be fed to the controller in HDR Ternary mode.

 **Note**

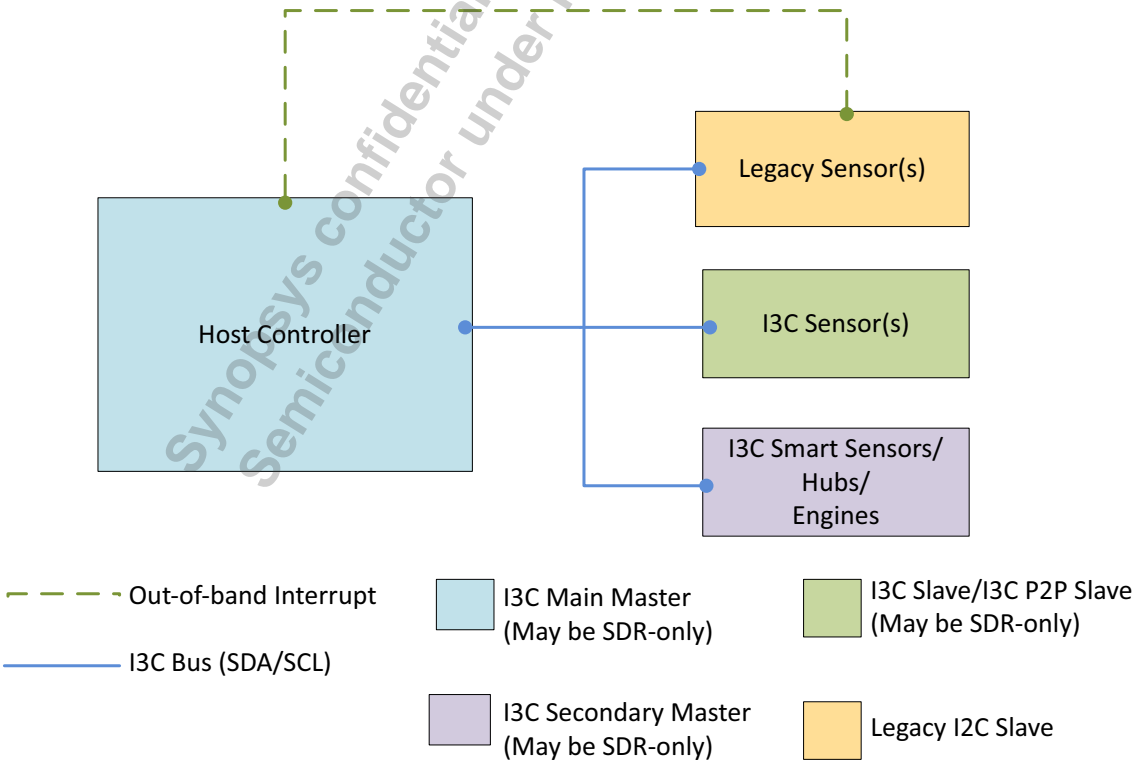
Ternary mode is not supported in this release.

### 1.1.1 System Block Diagram

The I3C system ensures reliable transmission and reception of data. The I3C interface is intended to improve upon the features of the I2C interface, while preserving backward compatibility with some limitations as mentioned in the MIPI I3C specifications 1.0 (for example, clock stretching is not supported).

Figure 1-1 shows the block diagram of I3C System.

Figure 1-1 I3C System





The I3C Protocol supports the following different roles of devices:

- Main Master
- Secondary Masters
- I3C Slaves
- I2C Slaves

The DWC MIPI I3C Slave-Lite Controller is a specialized I3C Slave that meets the requirement of the processor-less slaves connecting to I3C and I2C bus interfaces.

### 1.1.2 Applications

The Slave-Lite configuration is suitable for applications that need to connect registers with the I3C bus without requiring any application clock (excluding ternary mode of operation). For example, Slave-Lite is used to interface with external registers that capture the sensor data from sensors with I3C bus.

Typical applications built with Slave-Lite are Mobile SoC and other processor-less systems that target applications such as:

- Smart Phone
- Camera
- Touch Screen
- Wearables
- Memory interfaced applications
- Automotive, and so on

The following are some of the typical sensor classes addressed by I3C, and therefore can be built with Slave-Lite controller:

- Motion Sensors
  - Gyro
  - Touchscreen
  - Grip
  - Time of Flight (sensors)
- Biometric Sensors
  - Fingerprint
  - Heart Rate
- Environmental Sensors
  - Ambient Light
  - Temperature
- NFC (Near Field Communication)

### 1.1.3 Standards Compliance

Slave-Lite conforms to the following standards:

- MIPI® Alliance Specification for Improved Integrated Circuit (I3C<sup>SM</sup>), Version 1.0 Dec-2016

- JEDEC Module Sideband Bus Specification v0.53\_AA, July 2019, here after referred as JESD403-1

Synopsys confidential document, provided to GigaDevice  
Semiconductor under NDA, do not redistribute

## 1.2 Slave-Lite Features

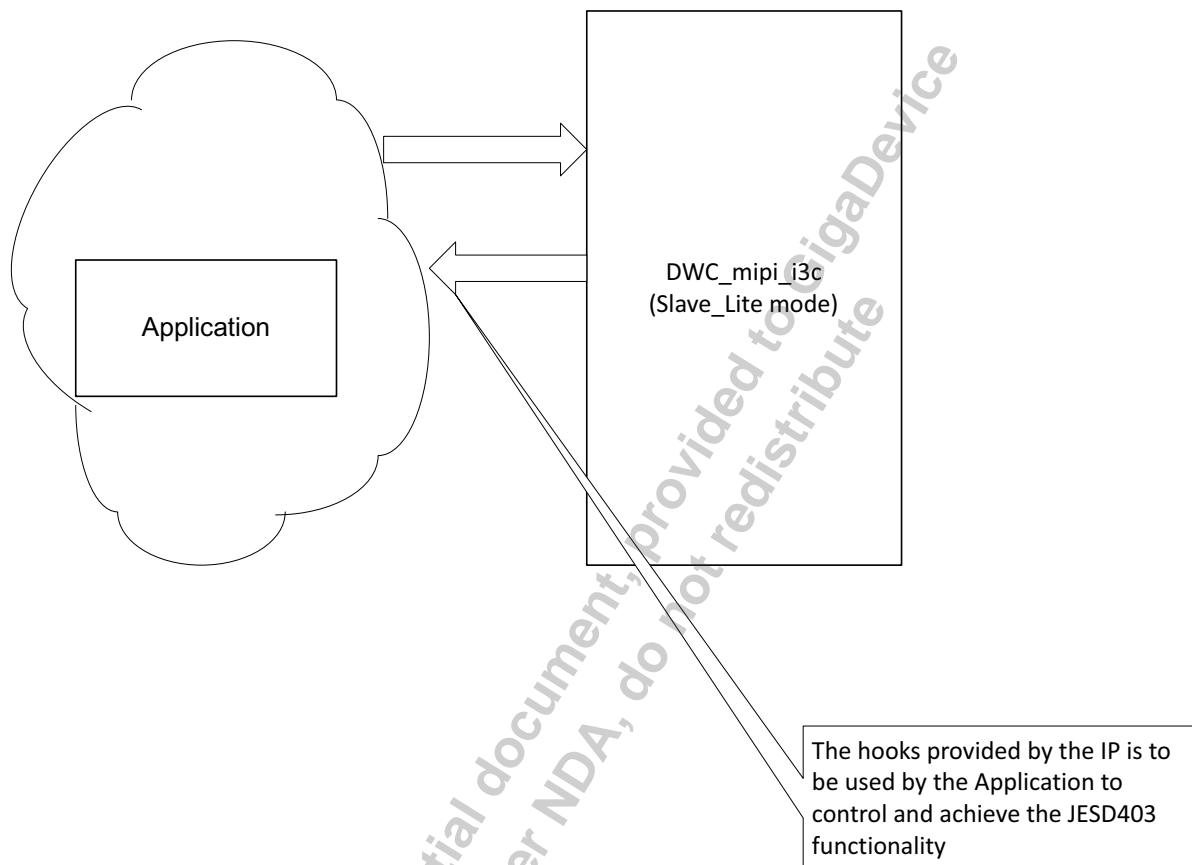
### 1.2.1 General Features

The Slave-Lite supports the following general features:

- Two Wire I3C serial interface - consists of a serial data line (SDA) and a serial clock (SCL)
- Does not require the application to provide a clock (applicable for SDR and HDR DDR modes only)
- Static or Dynamic Slave Device Support
- Built-in Hardware Dynamic Address Allocation support (ENTDAA/SETDASA/SETAASA)
- Built-in CCC command handler for autonomously supported CCCs (applications intervention are not required for all mandatory CCCs supported by the IP as listed in later section of this document.)
- Exclusive application interface for CCCs that are not handled by the IP including Vendor Specific CCCs.
- Support for Vendor Specific CCCs with defining byte support.
- Hot-Join Support
- Slave Interrupt Request Support with Mandatory Byte and payload data
- Automatic Retry mechanism for Arbitration loss (ENTDAA/IBI).
- Supported Data Rates:
  - FM/FM+/SDR/HDR-DDR
- Selectable sub-address size
- Parity Error/CRC Error Detection/Frame Error Detection
- I2C mode support
- Adaptive mode of operation between I2C and I3C depending on I3C bus traffic
- Configurable private data transfer interface width

#### 1.2.1.1 JESD403-1 Compliance in DWC\_mipi\_i3c in Slave-Lite Mode

The Slave-Lite mode is chosen to support JESD403-1 features because JESD403-1 compliant Slaves are closely tied to its register set on its application interface (like any sensor devices). The Slave devices are expected to return the register data for CCC and private read transfers (if supported) without issuing a NACK response, and hence DWC\_mipi\_i3c controller in Slave-Lite mode is most suitable for JESD403-1 features.

**Figure 1-2 Overview Of Application Interface**

While DWC\_mipi\_i3c in Slave-Lite mode is designed to support mandatory features of JESD403-1 and provide hooks to be used by the application to achieve the JESD403-1 functionality, the features/interface for JESD403-1 compliance is not supported in the current release. Contact Synopsys for further details. Eventually, DWC\_mipi\_i3c support for JESD403-1 for Slave-Lite configuration will be limited to two configurations that are representative of the JEDEC use case and can be found under “Validated Configurations” section of *DWC\_mipi\_i3c Controller User Guide*.

The application interface should support the synchronizers required to drive the interface of the IP which runs on SCL clock. The JESD403-1 specific registers are expected to be in the application and all read and write operations are performed with appropriate interface signals.

### 1.2.1.2 Bus Configuration

Slave-Lite can be used in the following three types of I3C Bus configuration as defined by the MIPI I3C Specification:

- Pure Bus: The configuration with only I3C devices present on the bus.
- Mixed Fast Bus: The configuration with both I3C devices and Legacy I2C slaves present on the bus. In this case, the Legacy I2C slaves are restricted to the ones that are generally permissible (that is, Slave-only, no clock stretching and that have a true I2C 50 ns glitch filter on SCL).

- **Mixed Slow/Limited Bus:** The configuration with both I3C Devices and Legacy I2C Slaves present on the Bus. In this case, the Legacy I2C Slaves are restricted to the ones that are selectively backward compatible with the I2C standard (that is, Slave-only and no clock stretching); but, these do not have a true I2C 50 ns glitch filter on SCL.

In Mixed Bus, the maximum possible data rate with I3C devices depends on the compliance of the I2C Slave as defined by the I2C Specification. The maximum data rate as specified in I3C Specification is possible only if all I2C slaves have the 50 ns spike filter.

In the absence of spike filters or if the presence of filter is unknown, the maximum data rate is limited to only FM or FM+ even for I3C devices (as per the I3C Specification). The I2C Slave is also not allowed to extend the clock.

#### 1.2.1.3 Device Address

In I3C System, Static Address devices are the ones that are enhanced from being an I2C Slave to being an I3C compliant slave, capable of obtaining a Dynamic Address. The Dynamic Address devices are the ones that are developed for I3C system and they do not have any I2C Static Address.

Slave-Lite obtains the Dynamic Address from the Current Master after it is enabled either through ENTDA, SETDASA, or SETAASA mechanism.

#### 1.2.1.4 In-Band Interrupt (IBI)

The I3C protocol also gives a provision of supporting In-Band interrupts within the 2-wire interface, which reduces the device pin count and signal paths. The In-Band interrupt is initiated by the Slave-Lite to the current Master by generating START on the bus.

Additionally, I3C also uses In-Band Interrupt to generate Hot-Join request (to request the host to assign I3C Dynamic Address).

## 1.3 Deliverables

The Slave-Lite requires the Synopsys coreConsultant tool to install, unpack, and configure the controller. The license file required to install, unpack, and configure the Slave-Lite is delivered separately.

The Slave-Lite image includes the following:

- Verilog RTL source code
- Synthesis scripts for Synopsys Design Compiler
- Verification testbench and test configurations derived from the parameter choices made during controller configuration

Synopsys confidential document, provided to GnaDevice  
Semiconductor under NDA, do not redistribute

## 1.4 Unsupported Features

- HDR TSP/TSL
- Point To Point (I3C Minimal Bus)
- Optional Timing Control (Synchronous/ Asynchronous)

Synopsys confidential document, provided to GigaDevice  
Semiconductor under NDA, do not redistribute

Synopsys confidential document, provided to GigaDevice  
Semiconductor under NDA, do not redistribute



# 2

## Architecture

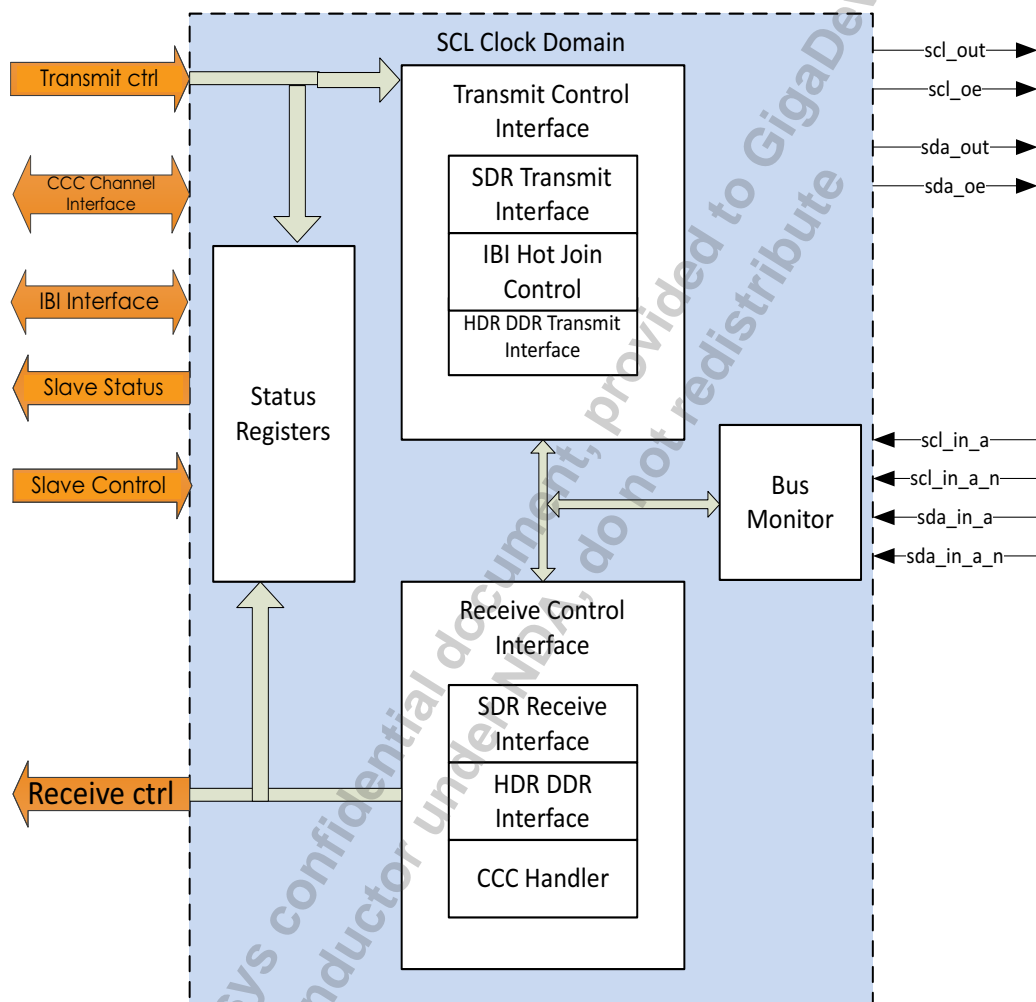
This chapter describes the Slave-Lite architecture. It contains the following sections:

- “Block Diagram of Slave-Lite” on page 26
- “Description of Slave-Lite Internal Blocks” on page 27
- “Description of Slave-Lite Bus Interfaces” on page 28
- “Functionality of Slave-Lite” on page 29
- “Sub-Address Generation” on page 30
- “Hot-Join Request Generation” on page 32
- “Slave Interrupt Request Generation” on page 35
- “Transfers” on page 44
- “Support for PEC in Private Transfers for JESD403-1 Compliance” on page 58
- “Error Handling” on page 61
- “I3C vs I2C Role Selection” on page 67
- “CCC Transfers with DWC MIPI I3C Slave-Lite Controller” on page 69
- “Support for Vendor Specific CCC with Defining Bytes” on page 72
- “Support for CCCs for JESD403-1 Compliance” on page 79
- “Handling of Unsupported CCC” on page 82
- “PEC Support for CCC for JESD403-1 Compliance” on page 83
- “Bus Reset” on page 86

## 2.1 Block Diagram of Slave-Lite

Figure 2-1 shows the block diagram of Slave-Lite.

**Figure 2-1 Block Diagram of Slave-Lite**



## 2.2 Description of Slave-Lite Internal Blocks

This section describes the internal blocks as shown in the Block Diagram (Figure 2-1).

### 2.2.1 Bus Monitor

The Bus Monitor module in Slave-Lite configuration detects the START, Re-START, and STOP conditions on the SDA line. It samples the SCL line status on the positive and negative edge of the incoming SDA to identify the STOP or START conditions. The STOP condition, when detected, is active only up to the point when the next START condition occurs.

### 2.2.2 Receive Control

The Receive Control Module samples the 8-bit incoming SDA value on the positive edge of the SCL to decode the following:

- Dynamic or Static address match based on the selection of `static_addr_en` input
- I3C Reserved Byte (7'h7E)
- I3C Common Command Code
- Sub-address field

This module handles the CCC information and programs the appropriate registers like dynamic address (DA), Maximum Write Length (MWL), Maximum Read Length (MRL), and so on.

### 2.2.3 Transmit Control

The transmit control module drives the slave responses for master requests on the bus. The transmit control module is responsible for transmitting the following:

- ACK/NACK for I3C Reserved (7'h7E) byte match
- ACK/NACK for Dynamic and Static address
- CCC Read Data
- Private Read Data
- Hot-join Request
- Slave In-band Interrupt

## 2.3 Description of Slave-Lite Bus Interfaces

The Slave-Lite controller consists of the following interfaces:

- Slave Control
- Slave Status
- IBI
- Private Read/Write Data
- HDR
- CCC Channel Interface



### Note

The Slave-Lite is targeted for processor-less designs where there might not be any application clock. Hence, the Slave-Lite controller outputs are all synchronous to Incoming SCL and the inputs are expected to be synchronous to the SCL. Because of these considerations, Slave-Lite does not implement any synchronization structure. If any application clock is present in the system, it is expected that the required synchronization for the inputs and outputs is implemented outside the controller.

For details about these signals, see [“Signal Descriptions”](#) on page 97.

## 2.4 Functionality of Slave-Lite

### 2.4.1 Overview of Slave-Lite Functionality

Slave-Lite is a processor-less Slave which runs on SCL clock generated by the Main Master controller on the I3C Bus. It supports features like sub-address generation which can be used for sensor applications.

Slave-Lite does not require programming.

### 2.4.2 Description of Slave-Lite Functionality

Slave-Lite can act as a static device or as a dynamic device depending on the `static_addr_en` input.

For the controller to act as a static device, the `static_addr_en` input must be asserted along with the static address provided on the `static_addr` input pins.

For the controller to act as a dynamic address device, both the `static_addr_en` and the `static_addr` pins must be tied to ground.

The controller, when configured as a static address device by asserting `static_addr_en`, responds to SETDASA CCC and ENTDAACCC to obtain its dynamic address. When configured as a dynamic address device, it responds to ENTDAACCC for obtaining the new dynamic address from the I3C Master.

## 2.5 Sub-Address Generation

### 2.5.1 Overview of Sub-Address Generation

The Sub-address field points to the register space in external applications (like sensors) or to FIFO from where the incoming data has to be written or read out.

The I3C Master always transacts with Slave-Lite by first writing the sub-address to which the intended read or write is supposed to happen, provided the sub-address field is required by the application.



#### Note

Sub-Addressing scheme is not supported in I2C mode of operation.

### 2.5.2 Description of Sub-Address Generation

The sub-address size can be selected by driving the `sub_addr_size` input pins in Slave-Lite. The `sub_addr_size` input can be driven by an external register maintained within the slave application.

**Table 2-1 Sub-address Bit Size**

sub_addr_size Input	Description
00	No sub-address is present. In this case, the <code>sub_addr_vld_t</code> output pin does not toggle and the value of the <code>sub_addr</code> output is invalid. This is provided if the external application does not require the sub-address field.
01	The first byte of the write transfer from the I3C Master is designated as the sub-address field.
10	The first two bytes of the write transfer from the I3C Master are designated as the sub-address field.
11	Reserved

If sub-address size is non-zero, the read or write transfers from the I3C Master is possible if a valid sub-address is received. The first `sub_addr_size` bytes of all write transfers is interpreted as sub-address field. For HDR mode of operation the sub address size must be 10.

This section explains:

- [“Writing to a Sub-Address”](#) on page 30
- [“Reading from a Sub-Address”](#) on page 31

#### 2.5.2.1 Writing to a Sub-Address

The following sequence describes writing to a sub-address:

- I3C Master initiates a write transfer by transmitting the Slave address.
- The first `sub_addr_size` of the payload data is considered as sub-address field.
- The valid sub-address, once received, is indicated to the application by toggling the output signal `sub_addr_vld_t`. The valid sub address is available at the output `sub_addr`.

- The bytes following the sub-address field is treated as payload data.
- After receiving each byte of payload data, the sub-address (`sub_addr`) is incremented to point to the next location. This change in sub address value is indicated to the application by toggling of the `sub_addr_vld_t` output.

### 2.5.2.2 Reading from a Sub-Address

The following sequence describes reading from a sub-address:

- I3C Master makes a write transfer to the Slave to write a new sub address from where data has to be read.
- The write transfer has a data length of `sub_addr_size` and ends with a Restart.
- The valid sub-address, once received, is indicated to the application by toggling the output signal `sub_addr_vld_t`. The valid sub address is available at the output `sub_addr`.
- I3C Master initiates a read transfer by transmitting the Slave address.
- To send each byte of payload data, it is indicated to the application by toggling `rd_data_rdy_t` output. The valid data should be available at the `rd_data` input.
- After sending each byte of read data, the sub address (`sub_addr`) is incremented to point to the next location. This change in sub-address value is indicated to the application by toggling of the `sub_addr_vld_t` output.

The timing diagrams in the sections [“Data Transfer with Sub-address”](#) on page 52 and [“Data Transfer without Sub-address”](#) on page 53 show the generation of the `sub_addr` and how it is incremented with every read and write transfer.

### 2.5.2.3 Parity Error During Sub-Address and Write Data Reception

When a parity error is encountered during sub-address reception, `sub_addr_vld_t` is not toggled. Error during this phase calls into question the validity of the sub-address to which subsequent data must be written, the controller stops detecting any further parity errors during the subsequent data bytes. The `wr_data_vld_t` does not toggle because of the received parity error, hence the incoming write data payload from Master is dropped. Thus, the application does not have any data to sample in this case.

When a parity error is detected after the sub-address phase (that is, sub-address is accepted successfully and the parity error occurred during the receipt of the payload data from Master), `wr_par_err_t` toggles and remains so upon the first occurrence of parity error. The controller stops further detection of parity errors in that transfer and resumes doing that after a Start/Restart/Stop (new transfer begins). In this case, `wr_data_valid_t` and `sub_addr_vld_t` continues to toggle for all bytes of data received for that transfer and it is up to the application to drop the incoming data packets based on the toggle of the `wr_par_err_t`.

The signal `wr_data_valid_t` can be used as the qualifier for sampling `wr_par_err_t`.

## 2.6 Hot-Join Request Generation

### 2.6.1 Overview of Hot-Join Request Generation

The application of the Slave-Lite can decide to send a Hot-join request to the current Master by asserting the Hot-join request signal `hj_req_now` after meeting the Bus Idle condition.

To meet the I3C Bus Idle condition, the application must monitor the `bus_idle` signal (output of the controller) to be high for at least  $t_{IDLE}$  period. The I3C Specification currently defines the minimum period of  $t_{IDLE}$  to be 1 ms.

### 2.6.2 Asserting `hj_req_now`

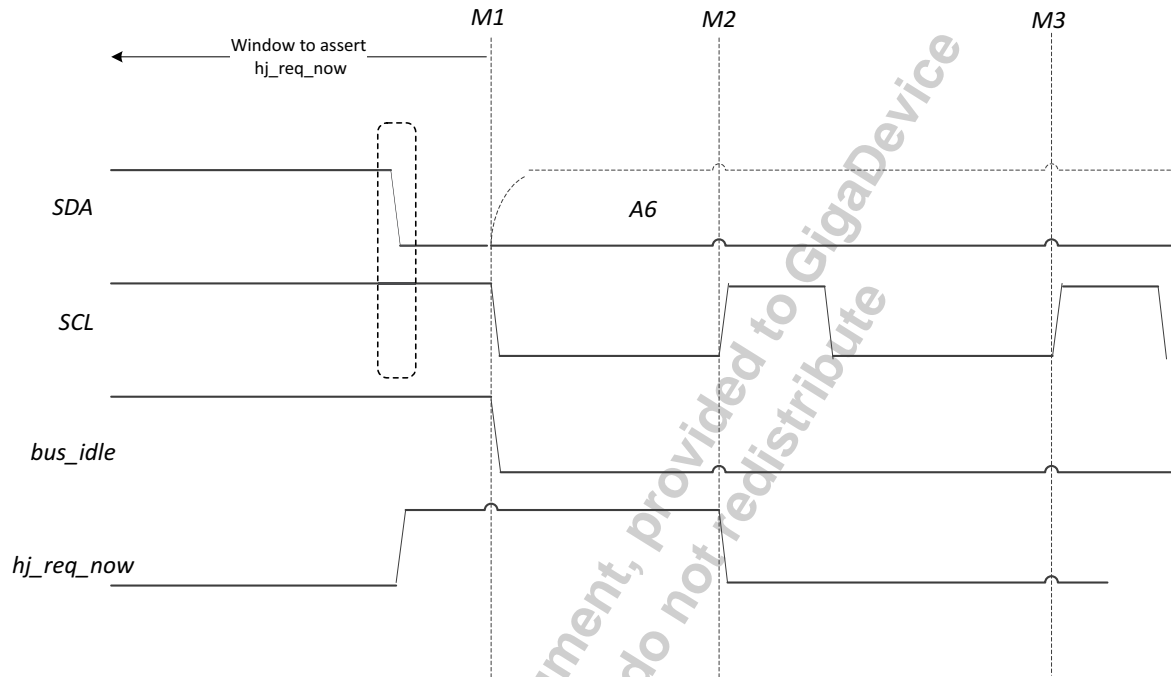
The Slave-Lite does not generate Hot-Join request automatically by monitoring the I3C bus. The application must have a counter that can detect a Bus Idle condition by monitoring the `bus_idle` signal. Since the SCL is not present during the Bus Idle condition, assertion of `hj_req_now` is asynchronous to SCL. `hj_req_now` must be asserted only when `bus_idle` is high. Figure 2-2 shows the window during which `hj_req_now` can be asserted.

Once the `hj_req_now` is asserted, the Slave directly pulls the SDA low and holds it until I3C Master starts providing the SCL. It is possible that I3C Master and Slave might want to generate START at the same time. Slave after detecting a Bus Idle condition asserts `hj_req_now` and at the same time I3C Master might have already generated a Start on the bus. In such scenarios, the Slave application must never assert `hj_req_now` after `bus_idle` goes low as indicated by Marker M1 in Figure 2-2. There can be an overlap in pulling SDA low by Master and Slave until  $t_{CAS}$  (clock after start), and Slave can assert `hj_req_now` until Master start providing the SCL (indicated by de-assertion of `bus_idle`). If the Slave application asserts `hj_req_now`, it should always happen before `bus_idle` goes low as indicated by the Marker M1.

### 2.6.3 De-asserting `hj_req_now`

De-assertion of `hj_req_now` must be always synchronous to SCL. Once asserted, `hj_req_now` must be high until next positive edge of SCL as indicated by Marker M2. The `hj_req_now` can be de-asserted synchronously with respect to any SCL edge before the next Stop condition on the bus. Once asserted, `hj_req_now` must be de-asserted before the next Stop condition on the bus.



**Figure 2-2 Hot-Join Request Generation**

The controller blocks the Hot-Join request from sending and returns with the appropriate status when one of the following condition is encountered:

- Hot-Join event generation is disabled by the current Master (through a DISEC CCC direct/broadcast command).
- The controller has already received the dynamic address.

For Hot-Join feature to be enabled, `slv_hj_ctrl` must be driven to 1 by the application. When `slv_hj_ctrl` is driven to 1, the controller does not participate in ENTDAAs until Hot-Join request is sent on the bus. When `slv_hj_ctrl` is driven to 0, Hot-Join feature gets disabled, and controller is not dependent on Hot-Join request to participate in ENTDAAs.

The controller sends a Hot-Join request if `hj_req_now` is asserted while Hot-Join event is enabled and dynamic address is not valid. Even if the controller loses the arbitration or the Master NACK's the Hot-Join request, controller keeps attempting the Hot-Join request on Start generated by Master until:

- Master ACK's the Hot-Join request
- Master assigns a dynamic address to the device
- Master disables the Hot-Join event by sending DISEC CCC



Application must not assert `hj_req_now` when Hot-Join event is disabled by the Master or when `slv_hj_ctrl` is driven to 0 or when dynamic address is valid. The output port:

- `slv_en_hj` indicates whether the slave Hot-Join event is enabled or disabled.
- `slv_dynamic_addr_vld` indicates whether the Slave dynamic address is valid or not.

The Status of the Hot-Join request is reflected in the `ibi_sts` port and is qualified by a toggle in `ibi_done_t` port. Table 2-2 describes the status information for Hot-Join request.

If the Hot-Join request by Slave loses arbitration or gets NACK'd, no status is given to the application. The Slave controller automatically generates the Hot-Join request on the next Master generated Start. The status is updated in the `ibi_sts` port only if the Master ACK's the Hot-Join request or if the Master disables the Hot-Join request, or if the Master assigns a dynamic address to the device.

**Table 2-2 Hot-Join Request Status Information**

<code>ibi_sts</code>	Value	Description
Success	01	ACK received for the HJ request from the current Master
Not Attempted	11	Controller has already got a valid Dynamic Address, or Hot-Join event is disabled by the current Master
Reserved	00	Default value
Reserved	10	Not used

The Slave application must always monitor the Bus Idle condition even after it has initiated a Hot-Join, until it receives a `ibi_sts` indicated by a toggle in `ibi_done_t`. If the Slave has initiated a Hot-Join, but lost the arbitration or Master NACK'd the request, the Slave application does not get any `ibi_sts`. The Slave retries the Hot-Join in the next Master generated Start without any trigger from the Slave application provided the Master generates a START. If the Master does not generate a START, the Slave application can assert the `hj_req_now` after the Bus Idle period for the controller to generate its own START by pulling the SDA low.

## 2.7 Slave Interrupt Request Generation

### 2.7.1 Overview of Slave Interrupt Request Generation

The application of the Slave-Lite can decide to send a Slave Interrupt Request by asserting either `sir_req_on_start` or `sir_req_now` request signals.

- The `sir_req_on_start` signal can be asserted any time and controller initiates an IBI request on next Start.
- The `sir_req_now` signal can be asserted only after meeting the Bus Available condition.

To meet the Bus Available condition, the application must monitor the `bus_idle` signal (output of the controller) to be high for at least  $t_{AVAIL}$  period. The I3C specification currently defines the minimum period to be 1 $\mu$ s.

### 2.7.2 Asserting `sir_req_now`

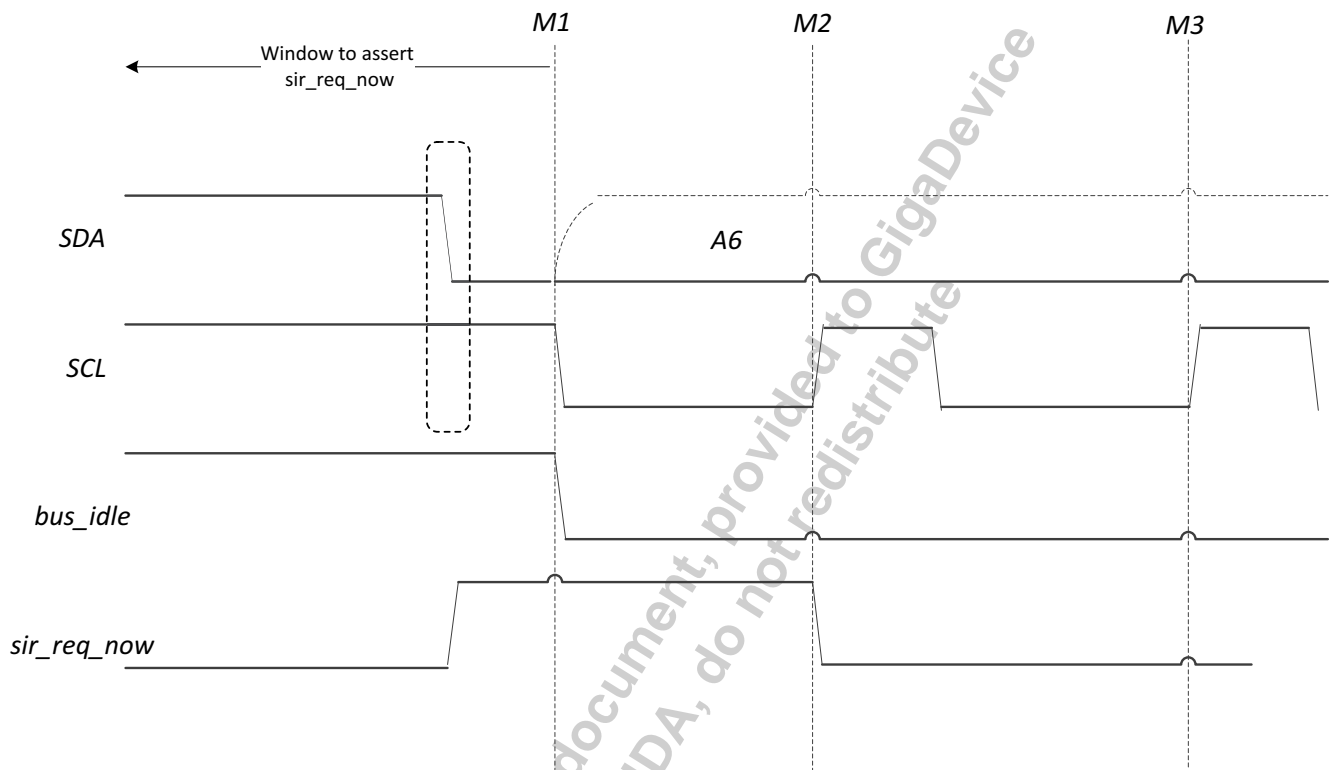
The `DWC_mipi_i3c_controller` does not generate Slave Interrupt Request (SIR) request automatically by monitoring the I3C bus. The application must have a counter that can detect a Bus Available condition by monitoring the `bus_idle` signal. As the SCL is not present during the Bus Available condition, assertion of `sir_req_now` is asynchronous to SCL. The `sir_req_now` signal must be asserted only when `bus_idle` is high. [Figure 2-3](#) shows the window during which `sir_req_now` can be asserted.

Once the `sir_req_now` is asserted, the slave directly pulls the SDA low and holds it until I3C Master starts providing the SCL. It can also happen that I3C Master and Slave wants to generate START at the same time. Slave, after detecting a Bus Available condition asserts `sir_req_now`, and at the same time I3C Master might have already generated a Start on the bus. In such scenarios, the Slave application should never assert `sir_req_now` after `bus_idle` goes low as indicated by Marker M1 in the [Figure 2-3](#).

There can be an overlap in pulling SDA low by Master and Slave until  $t_{CAS}$  (clock after start), and Slave can assert `sir_req_now` until Master start providing the SCL (indicated by de-assertion of `bus_idle`). If the Slave application asserts `sir_req_now`, it should always be done before `bus_idle` goes low as indicated by the Marker M1.

### 2.7.3 De-asserting `sir_req_now`

De-assertion of `sir_req_now` must be always synchronous to SCL. Once asserted `sir_req_now` must be high until next positive edge of SCL as indicated by Marker M2. The `sir_req_now` can be de-asserted synchronously with respect to any SCL edge before next Stop condition on the bus. Once asserted, `sir_req_now` should be de-asserted before next Stop condition on the bus.

**Figure 2-3 Assertion and De-assertion of sir\_req\_now**

#### 2.7.4 Asserting sir\_req\_on\_start

The signal `sir_req_on_start` can be asserted synchronous to SCL irrespective of the bus condition. The slave generates SIR request on the next Master generated Start condition.

#### 2.7.5 De-asserting sir\_req\_on\_start

Once asserted, the application is expected to keep the `sir_req_on_start` signal at high, until the controller responds back with the toggle on the `ibi_done_t` signal along with the `ibi_sts` status signal. The application must de-assert the `sir_req_on_start` within 1 SCL after toggling of `ibi_done_t`. The de-assertion of `sir_req_on_start` must also be synchronous to SCL.



#### Note

`sir_req_on_start` and `sir_req_now` must be asserted as specified in the “[Slave Interrupt Request Generation](#)” on page 35.

The controller blocks the SIR request from sending and returns with the appropriate status when the following conditions are encountered.

1. SIR event generation is disabled by the current master (through a DISEC CCC direct/broadcast command).
2. Dynamic address is not valid.

The controller sends a SIR request, if `sir_req_now/sir_req_on_start` is asserted while SIR event is enabled and dynamic address is valid. Even if the controller losses arbitration while sending SIR, the controller keeps re-attempting the SIR request until:

- Master ACKs/NACKs the SIR
- Master disables the SIR event by sending DISEC CCC
- Master resets the dynamic address through RSTDAA CCC

**Note**

The application should not assert `sir_req_now/sir_req_on_start` when the master disables SIR event, or when dynamic address is not valid. The output port `slv_en_int` indicates whether SIR event is enabled or disabled. The output port `slv_dynamic_addr_vld` indicates whether slave dynamic address is valid or not.

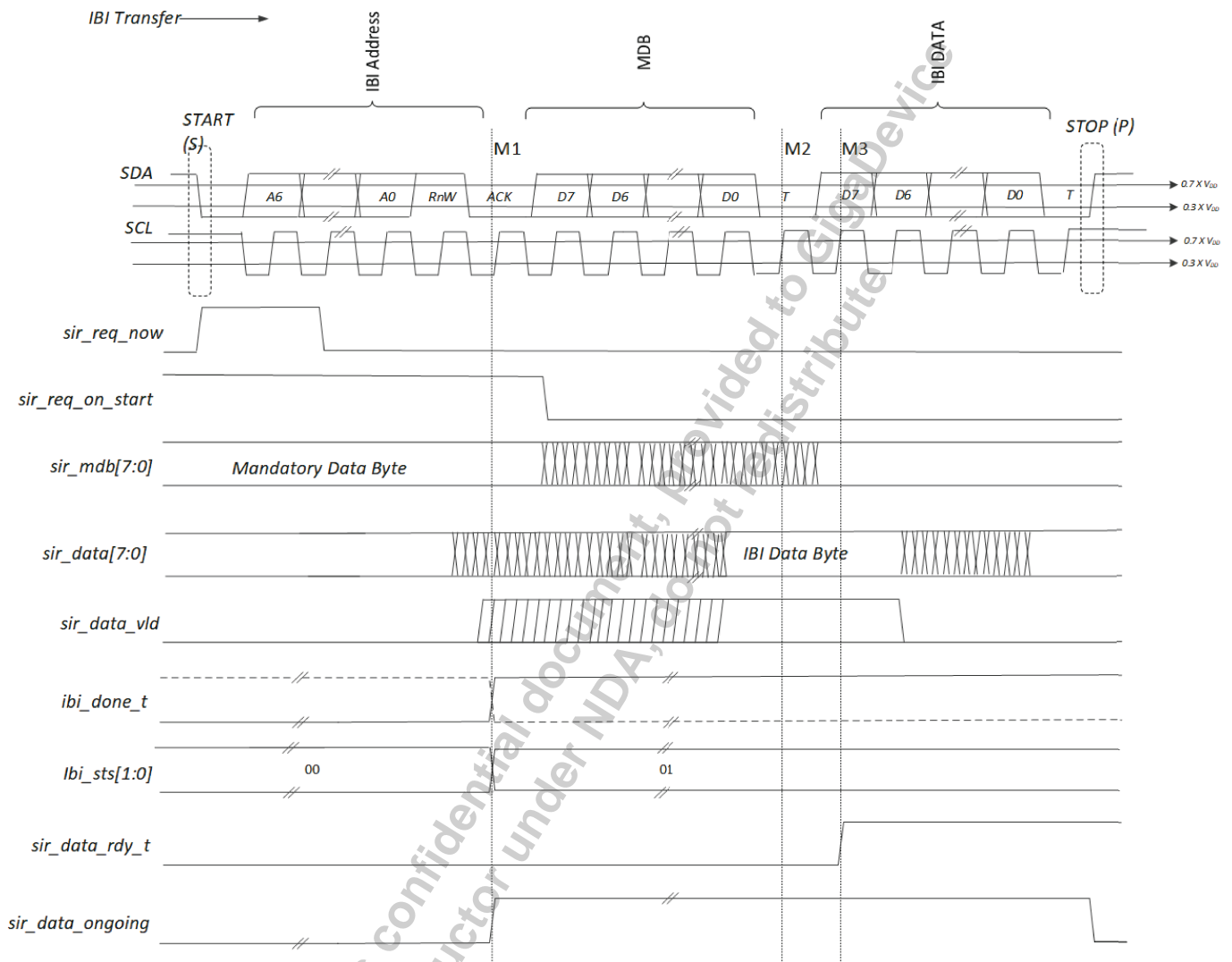
**Table 2-3 Description of IBI Status for Slave Interrupt Request**

ibi_sts	Value	Description
Success	01	ACK received for the SIR request from the Current Master
Failed	10	NACK received for the SIR request from the Current Master
Not Attempted	11	SIR event got disabled by the Current Master or Dynamic address is not valid
Reserved	00	Default Value

Slave application must monitor the Bus Available condition even after it has initiated a Slave Interrupt Request, until it receives an `ibi_sts` indicated by a toggle in `ibi_done_t`. If the Slave has initiated an SIR, and has lost the arbitration, the Slave application does not get any `ibi_sts`. The Slave re-tries the SIR in the next Master generated Start without any trigger from Slave application. But if Master did not generate a START, the Slave application can assert `sir_req_now` after Bus Available time for the controller to generate its own START by pulling the SDA low.

### 2.7.6 Slave Interrupt Request Generation with Data Payload

The Slave-lite controller supports sending data along with IBI using a custom IBI data interface. The data presented to the input ports `sir_mdb` and `sir_data` are sent to the Master if IBI request is ACK'ed. To enable IBI with data support, set the configuration parameter 'IC\_SLV\_IBI\_DATA'. This is also reflected in the BCR[2] register indicating that the Slave supports IBI payload. If BCR[2] bit in Slave is set, Master should accept at least the Mandatory Data Byte (MDB) if it decides to ACK the IBI. Both Master or Slave can terminate the IBI data during the T-bit of any data byte including MDB. The status of the IBI requested through 'sir\_req\_now' or 'sir\_req\_on\_start' is indicated in `ibi_sts[1:0]` port and the IBI status is qualified by the output signal `ibi_done_t`.

**Figure 2-4 IBI with Data and MDB**

If `ibi_sts[1:0]` indicates "01", it means that the Master has ACK'ed the IBI request. After the IBI is ACK'ed, the Slave sends the MDB. The Slave-Lite controller samples the `sir_mdb` at posedge of IBI ACK phase (Marker M1). The Slave-Lite application should ensure that `sir_mdb` input is valid before Marker M1. The controller samples the data present in the `sir_mdb` input port and sends as MDB if IBI is ACK'ed by the Master. The input port `sir_mdb` does not have any dependency on `sir_data_vld`. Also, `sir_data_rdy_t` does not toggle when the controller sends MDB. The `sir_data_ongoing` signal is set during the posedge of SCL if Master ACK's the IBI. The `sir_data_ongoing` signal indicates that the Slave controller sends the SIR data to the Master. The `sir_data_ongoing` is set during the MDB, and also during subsequent IBI data packets if present, until a restart or stop condition on the I3C bus.

The IBI data bytes after MDB are sampled from the `sir_data[7:0]` input port. The IBI data interface is similar to read data interface, having a valid signal for data and toggle signal indicating that data has been sampled. The `sir_data_vld` signal is the qualifier, which indicates that `sir_data[7:0]` is valid. The `sir_data_vld` should be valid before negedge of SCL during T-bit. The Slave controller decides whether to continue or to terminate the IBI transfer during the negedge of T-bit. If the Slave wants to continue IBI transfer, the Slave

drives 1 during the low period of T-bit, and if slave wants to terminate IBI transfer, it drives 0 during the low period of T-bit. The slave controller decides whether to continue or terminate an IBI transfer based on the `sir_data_vld` input during the negedge of T-bit. If `sir_data_vld` is high during negedge of T-bit, Slave continues the IBI transfer, and if `sir_data_vld` is low during negedge of T-bit, slave terminates the IBI transfer. Once asserted, `sir_data_vld` should be de-asserted only after toggling of `sir_data_rdy_t` signal.

The first byte after MDB is sampled on the posedge of T-bit of MDB as indicated by Marker M2, if `sir_data_vld` is high. `Sir_data_vld` is a valid signal for `sir_data`, and `sir_data` should be stable when `sir_data_vld` is high. The `sir_data` is sampled on the posedge of every T-bit until `sir_data_vld` is low during T-bit. Once `sir_data_vld` is low during T-bit, IBI transfer is terminated by the Slave. Whether the Slave controller has accepted the `sir_data` or not is indicated by the toggling of signal `sir_data_rdy_t`. The toggling of `sir_data_rdy_t` indicates that the controller has accepted the data provided in `sir_data` port. The `sir_data_rdy_t` toggles on the posedge of first bit every IBI data byte. That is, after the Slave has started sending the accepted `sir_data` to the Master. After the toggling of `sir_data_rdy_t`, the application can de-assert the `sir_data_vld` and change the `sir_data` value. The `sir_data_vld` and new `sir_data` should be provided again before the negedge of T-bit to continue the IBI transfer.

Master can also terminate the IBI transfer during the T-bit. Master can generate a restart during the high period of T-bit similar to Master terminate in read transfer. In Master terminate case, even though the Slave wants to continue IBI transfer, the Slave controller does not accept the `sir_data`. The `sir_data_rdy_t` does not toggle on the SCL edge after T-bit. After restart during the Master terminate, `sir_data_ongoing` is de-asserted indicating the application that IBI transfer has ended. The slave application can de-assert the `sir_data_vld` and change `sir_data` when `sir_data_ongoing` is de-asserted. The slave application can also differentiate that last `sir_data` provided to slave controller is not accepted as `sir_data_rdy_t` does not toggle.

### 2.7.6.1 SETMRL/GETMRL CCC Format

If IBI with data is supported, the Slave supports the 3-byte format for SETMRL/GETMRL CCC. Apart from supporting the 2-byte MRL value, the Slave also supports an additional byte for 'IBI Payload Size'. The output port `ibi_payload_size` is used to indicate the maximum IBI payload size. The default value of `ibi_payload_size` can be set from the configuration parameter- `IC_SLV_DFLT_IBI_PAYLOAD_SIZE`. The Master can read the IBI payload size using the GETMRL CCC. The slave returns the value in `'ibi_payload_size'` as 3rd byte of GETMRL CCC data. The Master can overwrite IBI payload size by sending the SETMRL CCC. This updated IBI payload size value is reflected in the `ibi_payload_size` output port.

The Slave controller is transparent to 'IBI Payload Size' value. It is the Slave applications responsibility to control the data length of IBI transfer appropriately depending the 'IBI Payload Size' value. The slave controller terminates an IBI transfer when `sir_data_vld` is not high during the negedge of T-bit and controller does not depend on IBI payload size set by Master. Upon receiving a SETMRL CCC, the Slave controller updates the new IBI payload size in the `ibi_payload_size` port and the slave application can take appropriate action. This behavior is similar to Slave controller behavior in read transfers. Slave controller does not restrict read data length based on MRL value set by Master and the responsibility of read termination is upon slave application.

### 2.7.6.2 PEC Support for IBI with Data for JESD403-1 Compliance

The controller also supports Packet Error Check (PEC) byte for IBI payload. The PEC is calculated on both Slave Address and IBI payload and is appended as last byte of IBI payload. If `'slv_pec_enable'` input port is driven to 1, the slave controller sends the PEC byte as the last byte of IBI payload. The PEC byte is automatically calculated by the Slave controller like PEC byte in case of private/CCC read transfers.



The PEC feature for JESD403-1 compliance is not supported in the current release. Eventually, DWC\_mipi\_i3c support for JESD403-1 for Slave-Lite configuration is limited to two configuration that are representative of the JESD403-1 use case and can be found under “Validated Configurations” section of DWC\_mipi\_i3c Controller User Guide.

**Figure 2-5 PEC Support for IBI with Data**

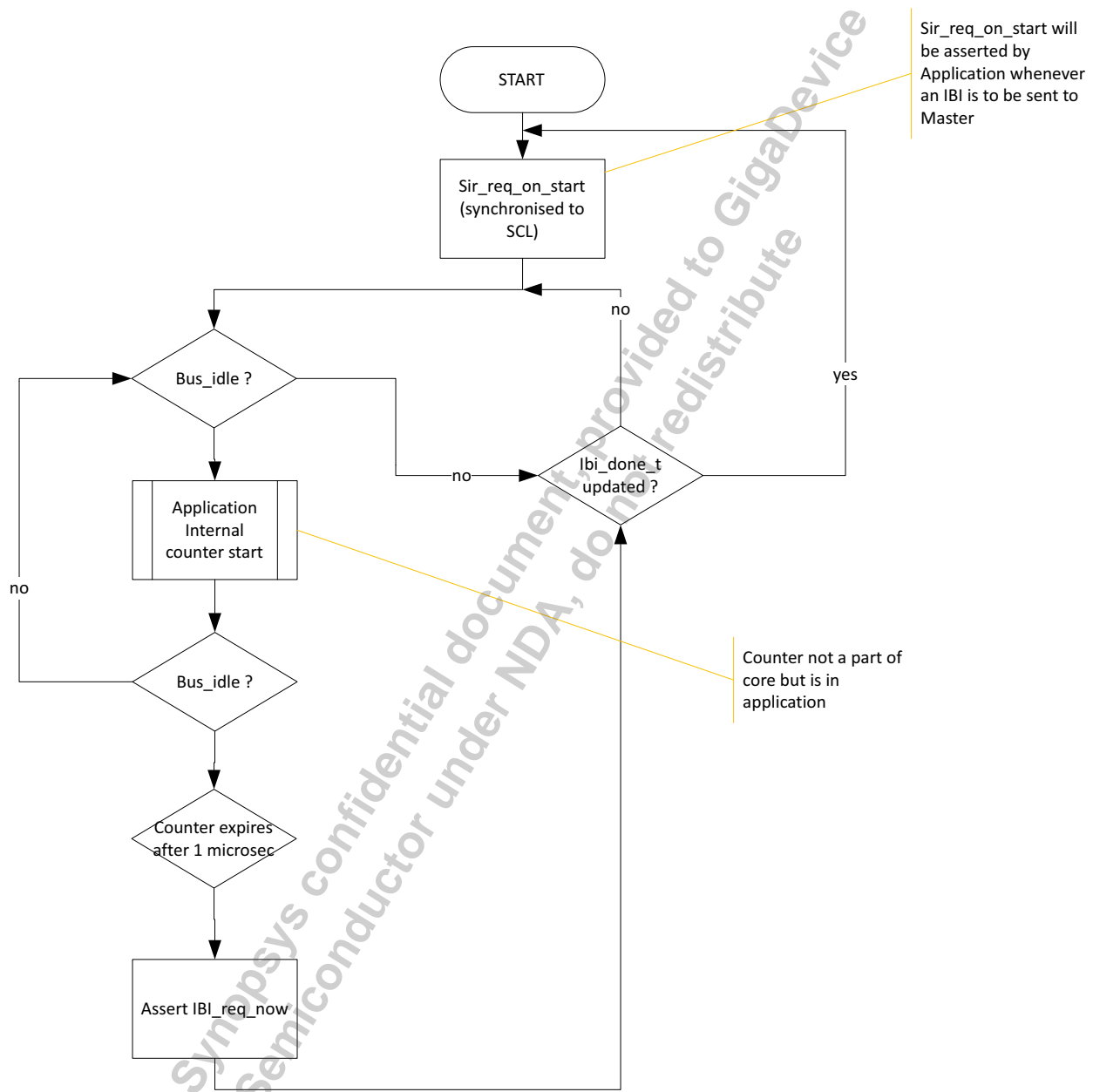
START (S)	Slave Address (SA,r)	IBI Payload (IBI DB,T)	PEC Byte (CRC Byte,T)	STOP (P/Sr)
--------------	-------------------------	---------------------------	--------------------------	----------------

### 2.7.7 Slave Interrupt Request Generation Flow

The signals `sir_req_now` and `sir_req_on_start` cannot be asserted independently. The assertion of the signal `sir_req_on_start` must always precede `sir_req_now` from the Slave application. The Slave application asserts the `sir_req_on_start` whenever an IBI is to be sent to the I3C Master. It is independent of I3C bus condition. After asserting the `sir_req_on_start` signal the application must assert the `sir_req_now` if the bus is idle of Bus Available Time.

Figure 2-6 depicts the flow for `sir_req_on_start` and `sir_req_now` assertion



**Figure 2-6** sir\_req\_on\_start and sir\_req\_now Assertion Flow

Whenever the Slave application wants to send an SIR request, assert sir\_req\_on\_start.

If the bus is not idle, wait for ibi\_done\_t. Once ibi\_done\_t is received de-assert sir\_req\_on\_start.

If the bus is idle, count for Bus Available time.

If Bus Available time is expired, assert sir\_req\_now. If the bus becomes non-idle before Bus Available expiry, wait for ibi\_done\_t.

## 2.7.8 SIR Generation when Bus is not Idle

Figure 2-7 shows SIR request generation by asserting `sir_req_on_start`. When the bus is not idle `sir_req_on_start` is asserted synchronous to SCL. The Slave sends SIR request on next Start generated by the Master. After the Master ACK's, NACK's the `ibi_done_t` toggles indicating the IBI status in `ibi_sts` port. De-assertion of the `sir_req_on_start` must be done within one SCL clock period after `ibi_done_t` toggles.

Figure 2-7 SIR Request Generation by Asserting `sir_req_on_start`

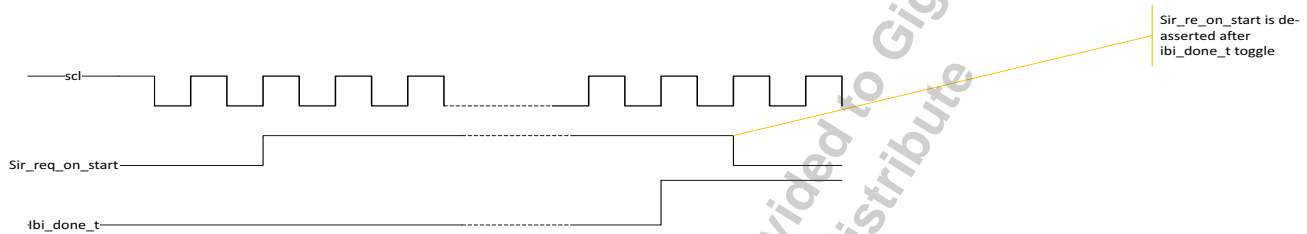
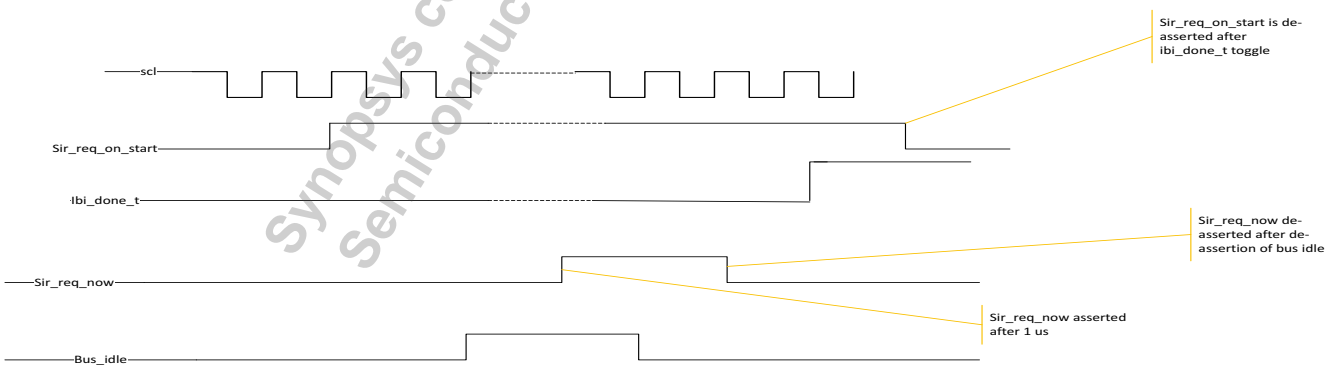


Figure 2-8 shows SIR request generation by asserting `sir_req_now` and `sir_req_on_start`. When the bus is not idle `sir_req_on_start` is asserted synchronous to SCL. The bus becomes idle after Master generates Stop. The Slave application starts the Bus Available counter since the bus is idle. Once the Bus Available count expires:

- Application generates `sir_req_now`
- Slave immediately pulls SDA low to send SIR request
- The Master starts providing SCL for Slave to send SIR request

The signal `sir_req_now` is de-asserted synchronous to SCL once `bus_idle` is de-asserted. After the Master ACK, NACK or Disable `ibi_done_t` toggles indicating the IBI status in `ibi_sts` port. De-assert the `sir_req_on_start` within 1 SCL after `ibi_done_t` toggle.

Figure 2-8 SIR Request Generation by Asserting `sir_req_now` and `sir_req_on_start` (When Bus is not Idle)

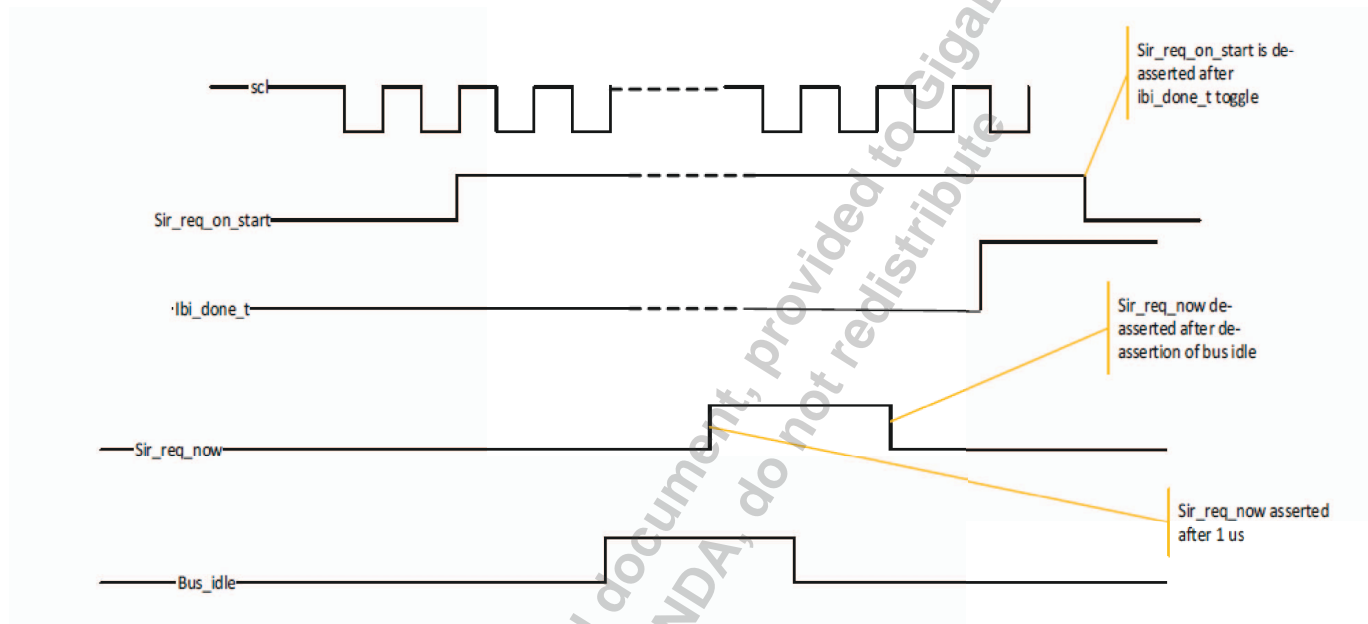


## 2.7.9 SIR Generation when Bus is Idle

Figure 2-9 shows the SIR request generation by asserting `sir_req_now` and `sir_req_on_start`. The `sir_req_on_start` signal generated from the application reaches the Slave interface only if SCL is present. As the bus is idle, the Slave application starts the Bus Available counter. Once the Bus Available count expires,

the application generates `sir_req_now`. The slave immediately pulls SDA low to send the SIR request. The Master starts providing SCL for Slave to send SIR request. The `sir_req_on_start` signal reaches the Slave interface once SCL is available. `sir_req_now` is de-asserted synchronous to SCL once `bus_idle` is de-asserted. After the Master ACK, NACK, or Disable, `ibi_done_t` toggles indicating the IBI status in `ibi_sts` port. De-assert the `sir_req_on_start` within one SCL after `ibi_done_t` toggle.

**Figure 2-9 SIR Request Generation by Asserting `sir_req_now` and `sir_req_on_start` (When Bus is Idle)**



## 2.8 Transfers

### 2.8.1 Introduction to Private Data Transfers

The Slave-Lite configuration supports private read and write transfers through simple private data interface.

The interface width can be configured as:

1. 8 bits for transmit receive channel (rd\_data = 8 bits and wr\_data = 8 bits)
2. 32 bits for transmit channel (rd\_data = 32 bits) and 16 bits for receive channel (wr\_data = 16 bits)

#### 8 bit Interface:

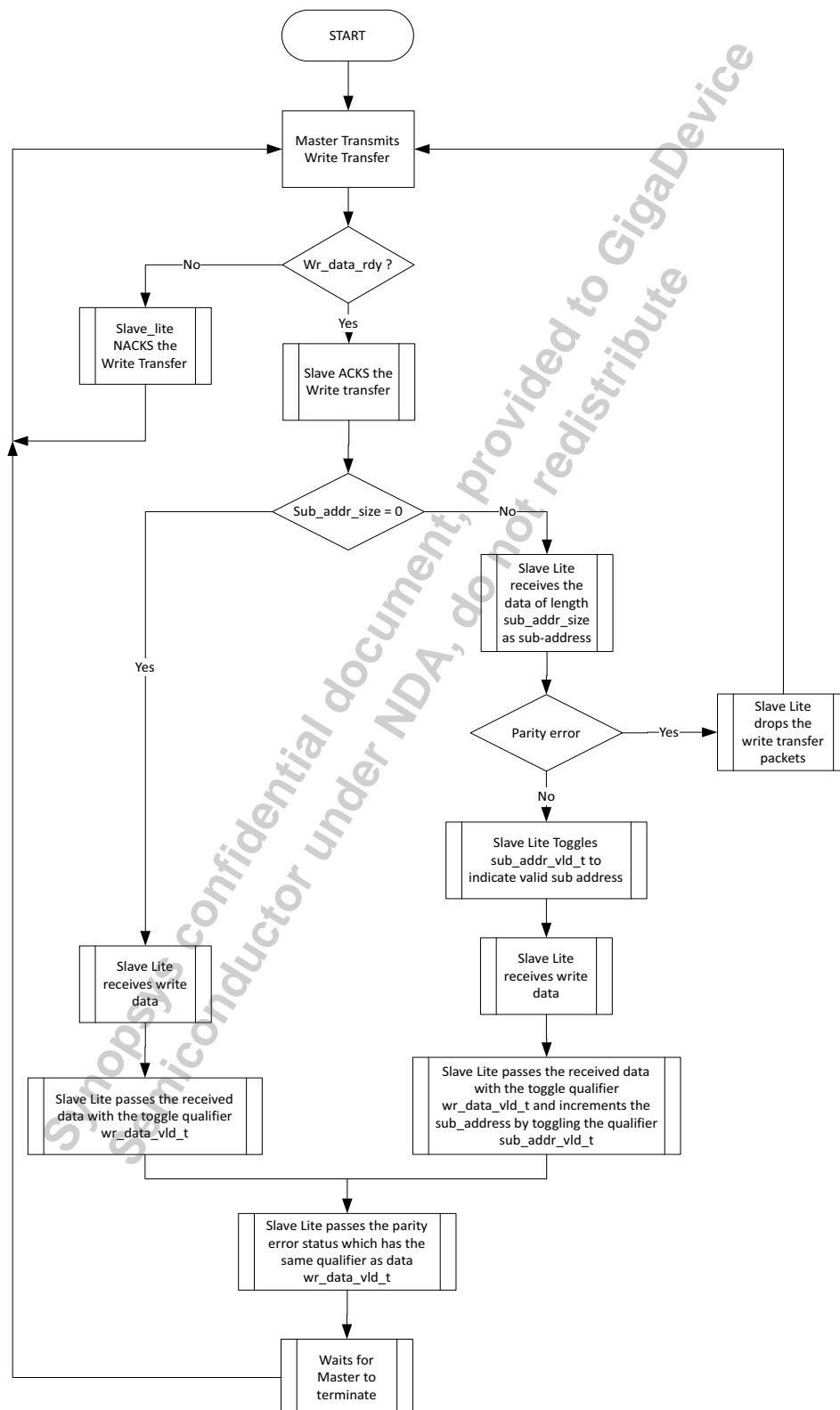
The 8 bit interface is applicable only for SDR mode of operation. For HDR mode of operation, 32/16 bit interface must be configured. The 8 bit data of transmit channel (rd\_data) is qualified by the rd\_data\_vld signal from the Slave application. Similarly, the 8 bit data from the receive channel (wr\_data) is qualified by wr\_data\_vld\_t.

#### 32/16 bit Interface:

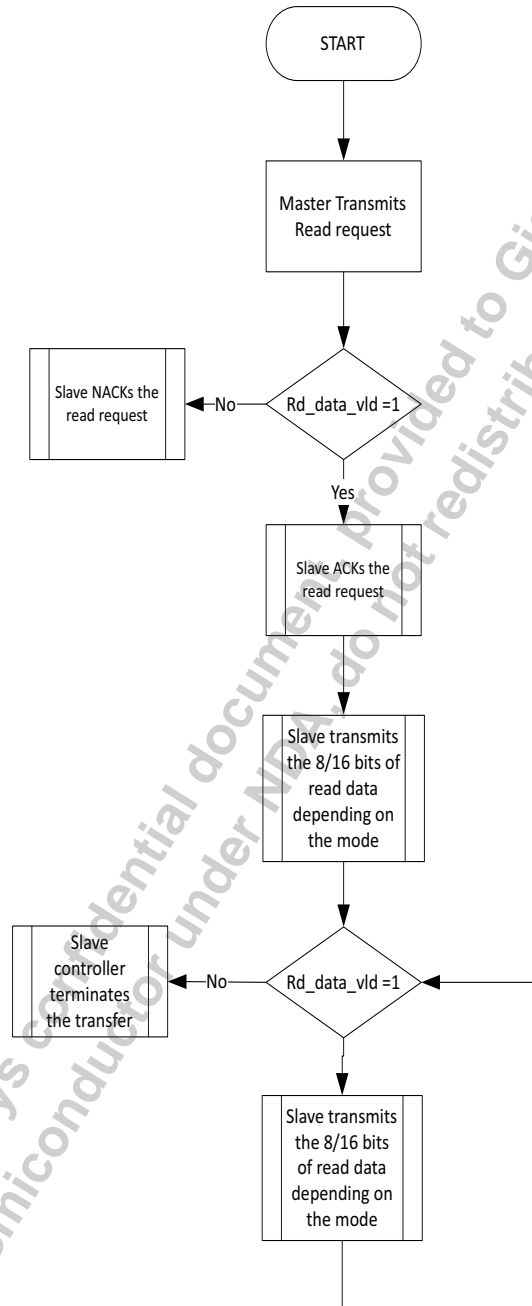
The 32/16 bit interface can be configured for both SDR and HDR mode of operations. The 32 bit data of transmit channel (rd\_data) has two bits enabled to delimit the valid bytes in the last transfer, which might not be in the boundary of 32 bits. The rd\_data\_vld is used as the qualifier for the 32 bits rd\_data.

The 16 bit receive channel (wr\_data) has a single bit enabled, which is set during the HDR mode of transfer indicating all 16 bits of wr\_data are valid. In SDR mode of operation, the bit is not set indicating validity of only 8 bits of data. The 32/16 bit interface requires less frequent updating of the transmit data for Master Read transfers. Similarly, less frequent sampling of the received data is required for Master Write transfers in HDR mode.

[Figure 2-10](#) explains the flow of the write and read transfers.

**Figure 2-10 Private Write Transfer Flow**


- `Wr_data_rdy` signal is the qualifier for the Slave-Lite to accept the Master Write Transfer. If the signal is not asserted, the controller NACK's the write request from the I3C Master.
- If a valid value of `sub_address_size` is present, the controller considers the first "`sub_addr_size`" of the payload as sub address field else the controller considers the payload as valid data from the I3C Master.
- If a Parity error is detected during sub-address phase, the sub address qualifier does not toggle and the following data is dropped.
- The payload data after the sub address is treated as valid write data and is qualified with the toggle signal `wr_data_vld_t` and passed to the Slave-Lite application.
- If the Master write interface is selected as 16 bits, the data is associated with a byte enable. For SDR mode of operation, the byte enable is '0' which indicates that only the Least Significant Byte of the 16 bits of data is valid. In HDR mode of operation, the byte enable is '1' which indicates that all the 16 bits of data are valid.
- For Parity error during receiving of the write data, the Slave-Lite application must discard the incoming data by ignoring the write data valid until the Master terminates the transfer.

**Figure 2-11 Private Read Transfer Flow**

The Slave-Lite expects to have the read data readily available at the time the READ Transfer request is received from the Master. The availability of the read data is indicated by the `rd_data_vld` signal. Slave-Lite NACKs the read transfer when `rd_data_vld` is not present.

The Slave-Lite generates a toggle signal `rd_data_rdy_t` whenever the complete data from the application is latched internally. After sending the data, Controller checks for the `rd_data_vld` signal to understand whether it has to terminate the transfer or continue with next set of data.

**Note**

In [Figure 2-12](#) to [Figure 2-19](#), the exact point of toggle of the signals can be determined by referring to the 'Synchronous to' field in ["I3C Interface Signals"](#) on page 100.

## 2.8.2 Private Write Transfers with Eight Bit Interface

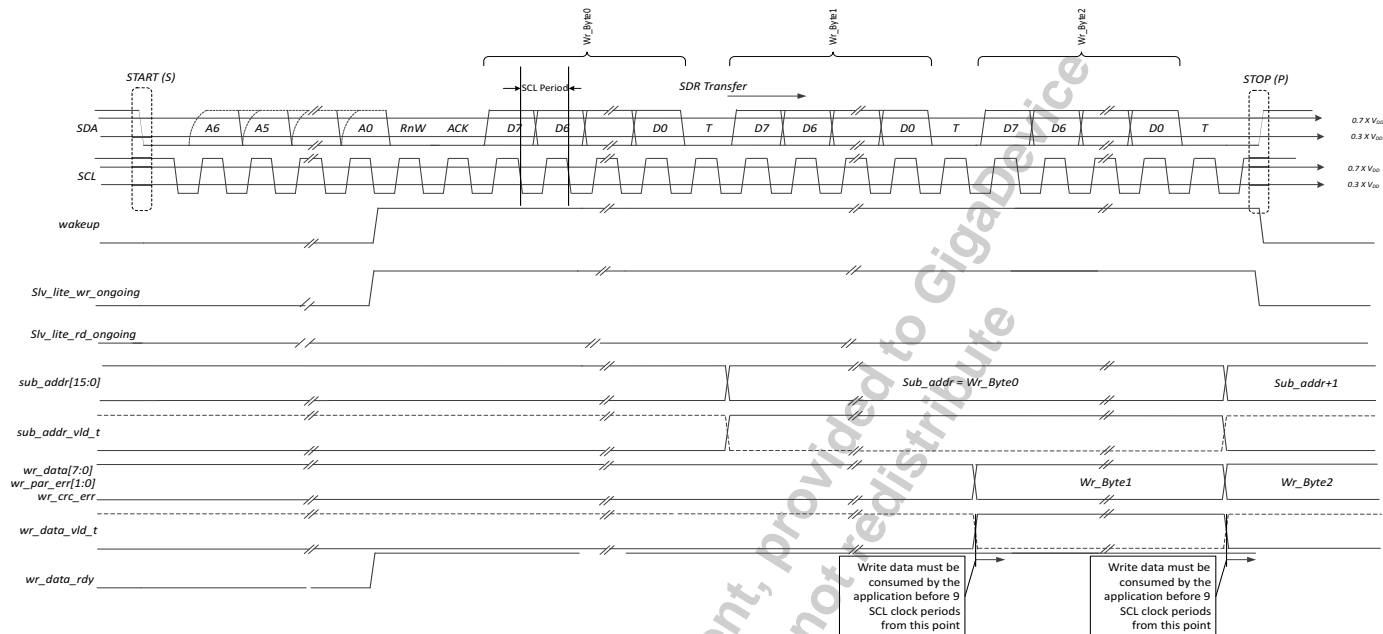
### 2.8.2.1 Write Transfer with Sub-address

The following signals represent Private Write Transfer with Sub-address:

- **Wakeup:** The wakeup signal indicates to the application that some transfer request from the Master is meant for this slave and it must be ready to receive or send the data for the request. It is asserted when the slave address matches with the incoming address on the line after START, and it is de-asserted when there is no address match after START or when a STOP is detected.
- **Slv\_lite\_wr\_ongoing:** This signal is asserted whenever there is an address match for the Master Write transfer targeted for this slave. This signal remains asserted until the write transfer is over from the Master.
- **Sub\_addr\_vld\_t:** This signal toggles if there is no parity error during the reception of the sub-address. During the write byte reception, it toggles along with the wr\_data\_vld\_t when the sub-address value increments.
- **wr\_par\_err\_t:** If there is any parity error, the wr\_par\_err\_t toggles and the qualifier for sampling it is wr\_data\_vld\_t.
- **wr\_data\_rdy:** This is the write data ready level indicator from the application to indicate the readiness for accepting the wr\_data. The controller is expected to see this signal high every time before the start of the write transfer. This input should be synchronized to the SCL clock domain.
- **wr\_data\_vld\_t:** This signal toggles whenever eight bits of data has been received and the parity is calculated. This is the qualifier for wr\_data and wr\_par\_err\_t.
  - **SDR mode:** The wr\_data should be sampled within eight SCL clocks after the wr\_data\_vld\_t toggles. If the SCL frequency is 12.5 MHz, the approximate time for eight SCL clocks is 640 ns.
- **wr\_crc\_err\_t:** This signal toggles only when CRC error is detected after the DDR data phase. Hence, this signal toggles when wr\_data\_vld\_t is not present and has to be sampled separately for determining the error.

[Figure 2-12](#) shows the timing diagram of Private Write Transfer with Sub-address.



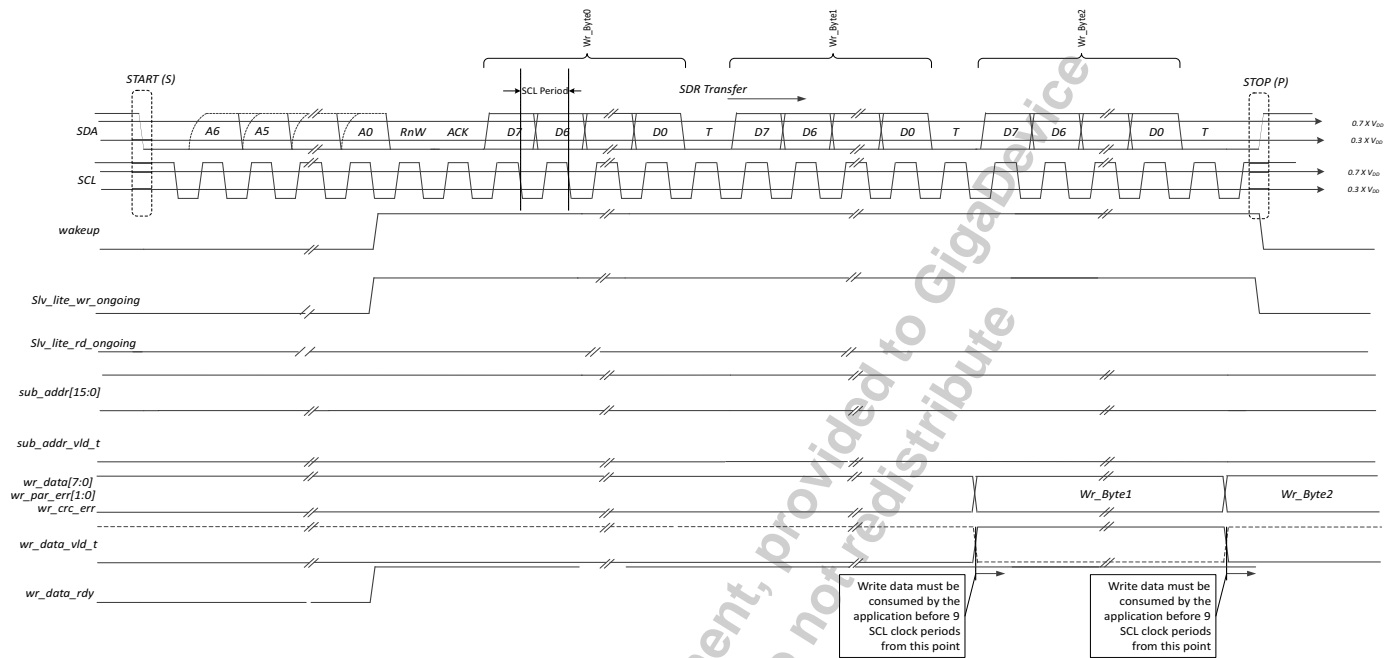
**Figure 2-12 Timing Diagram of Private Write Transfer with Sub-address**

### 2.8.2.2 Write Transfer without Sub-address

This mode can be selected by driving zero on input signal `sub_addr_size`. The following signals represent Private Write Transfer without Sub-address:

- **Wakeup:** The wakeup signal indicates to the application that some transfer request from the Master is meant for this slave and it must be ready to receive or send the data for the request.  
It is asserted when the slave address matches with the incoming address on the line after START, and de-asserted when there is no address match after START or when a STOP is detected.
- **Slv\_lite\_wr\_ongoing:** This signal is asserted whenever there is an address match for the Master Write transfer targeted for this slave. This signal remains asserted until the write transfer is over from the Master.
- **wr\_par\_err\_t:** If there is any parity error, the `wr_par_err_t` toggles and the qualifier for sampling it is `wr_data_vld_t`.
- **wr\_data\_rdy:** The write data ready level indicator from the application to indicate the readiness for accepting the `wr_data`. The controller is expected to see this signal high every time before the start of the write transfer. This input should be synchronized to the SCL clock domain.
- **wr\_data\_vld\_t:** This signal toggle whenever 8 bits of data has been received and the parity is calculated. This is the qualifier for `wr_data` and `wr_par_err_t` or `wr_crc_err`. The `wr_data` should be sampled within the stipulated time interval depending on the mode of operation:
  - **SDR mode:** The `wr_data` should be sampled within 8 SCL clocks after the `wr_data_vld_t` toggles. If the SCL frequency is 12.5 MHz the approximate time for 8 SCL clocks is 640 ns

Figure 2-13 shows the timing diagram for Private Write Transfer without Sub-address.

**Figure 2-13 Timing Diagram for Private Write Transfer without Sub-address**

## 2.8.3 Private Write Transfers with Sixteen Bit Interface

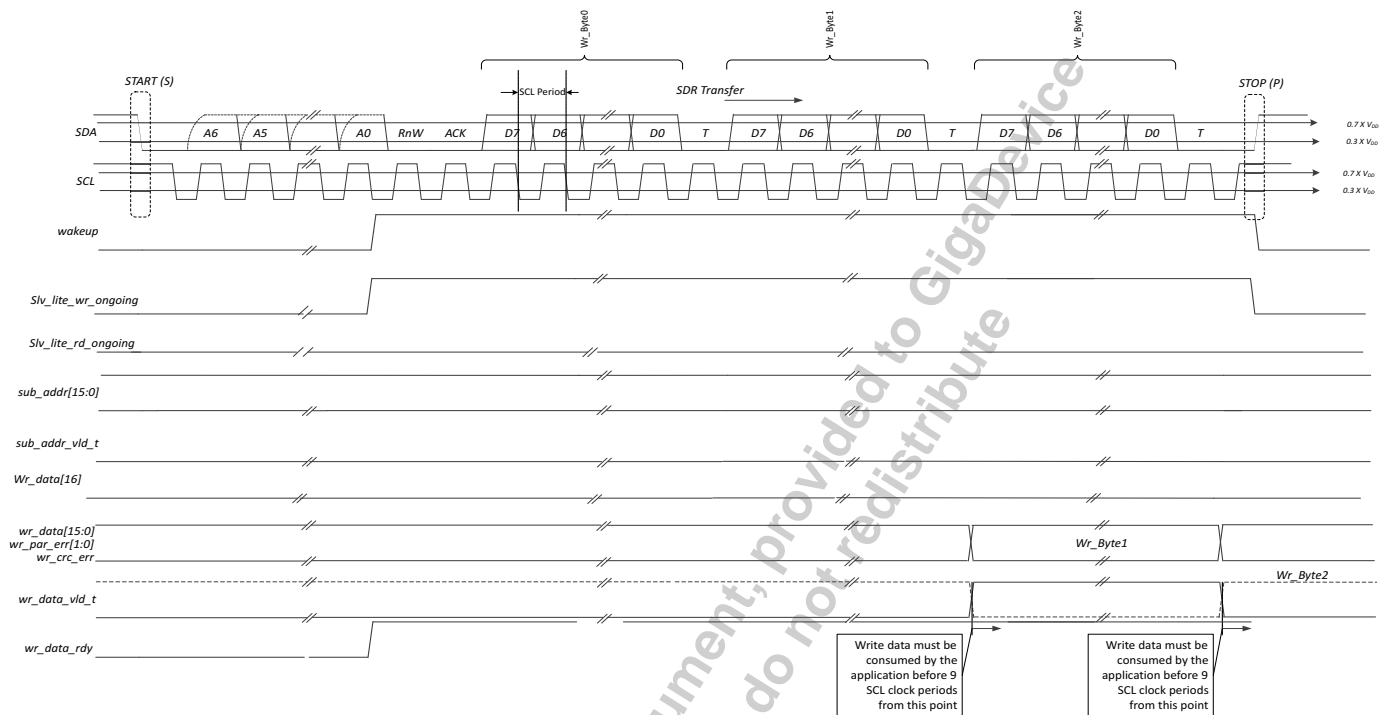
### 2.8.3.1 SDR Mode with Byte Enable without Sub-address

The wr\_data interface width is of 17 bits, where the most significant bit is to be used as a byte enable and the rest of the 16 bits are to be used as data. In the SDR mode of operation only the lower 8 bits of the wr\_data signal is used. The byte enable bit (wr\_data[16]) is always zero, indicating out of the 16 bits of data only the lower 8 bit is valid.

The following signals represent Private Write Transfer in SDR mode:

- **Wakeup:** The wakeup signal indicates to the application that some transfer request from the Master is meant for this slave and it must be ready to receive or send the data for the request. It is asserted when after START the slave address matches with the incoming address on the line, and de-asserted when after START there is no address match or when a STOP is detected.
- **Slv\_lite\_wr\_ongoing:** This signal is asserted whenever there is an address match for the Master Write transfer targeted for this slave. This signal remains asserted until the write transfer is over from the Master.
- **wr\_par\_err\_t:** If there is any parity error, the wr\_par\_err\_t toggles and the qualifier for sampling it is wr\_data\_vld\_t.
- **wr\_data\_rdy:** The write data ready level indicator from the application to indicate the readiness for accepting the wr\_data. The controller is expected to see this signal high every time before the start of the write transfer. This input should be synchronized to the SCL clock domain.

Figure 2-14 shows the timing diagram for Private Write Transfer in SDR Mode.

**Figure 2-14 Timing Diagram for Private Write Transfer in SDR Mode**

### 2.8.3.2 HDR-DDR Mode with Byte Enable and with Sub-address

The `wr_data` interface width is of 17 bits, where the most significant bit is to be used as a byte enable and the rest of the 16 bits are to be used as data. In HDR DDR mode of operation all the bits of the `wr_data` signal is used. The byte enable bit (`wr_data[16]`) is always one, indicating all the 16 bits of data are valid.

The following signals represent the Private Write Transfer in HDR-DDR Mode:

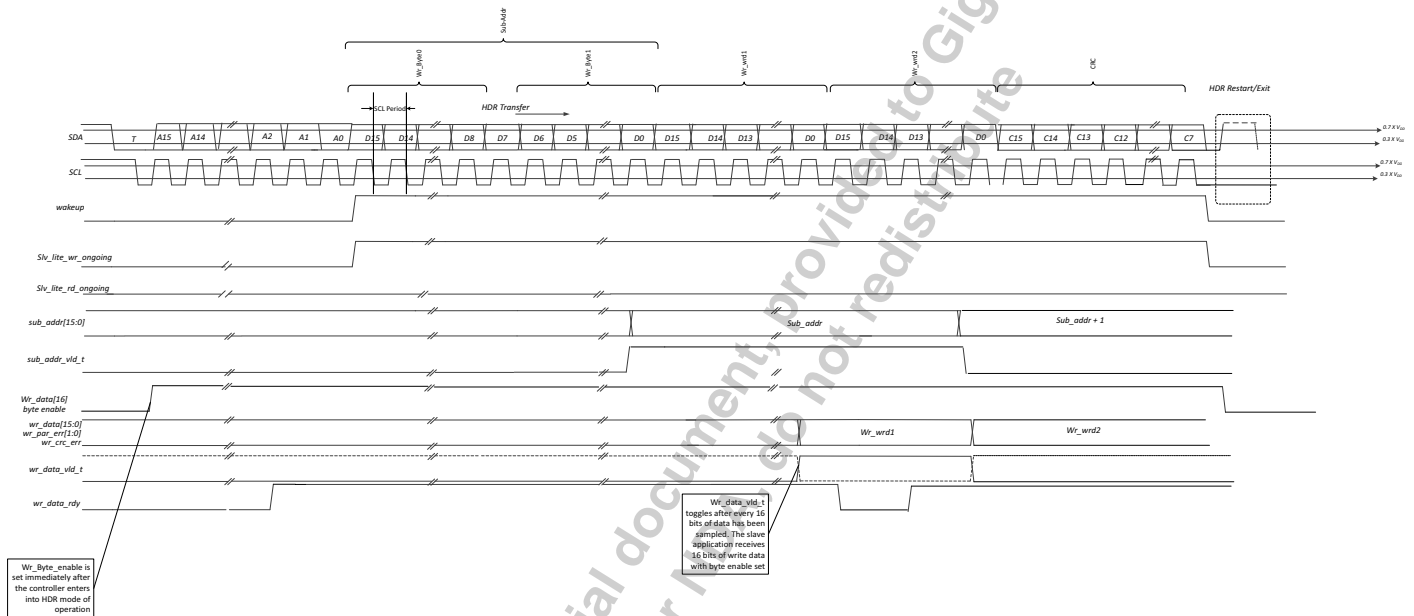
- **Wakeup:** The wakeup signal indicates to the application that some transfer request from the Master is meant for this slave and it must be ready to receive or send the data for the request. It is asserted when after START the slave address matches with the incoming address on the line, and de-asserted when after START there is no address match or when a STOP is detected.
- **Slv\_lite\_wr\_ongoing:** This signal is asserted whenever there is an address match for the Master Write transfer targeted for this slave. This signal remains asserted until the write transfer is over from the Master.
- **Sub\_addr\_vld\_t, sub\_addr:** The first two bytes sent by Master are treated as the sub-address. The sub-address size should be programmed to 2'b10. The sub-address increments after the first word of write data from the master is received and put into the sub-address location.
- **wr\_par\_err\_t:** If there is any parity error, the `wr_par_err_t` toggles and the qualifier for sampling it is `wr_data_vld_t`.
- **wr\_data\_rdy:** The write data ready level indicator from the application to indicate the readiness for accepting the `wr_data`. The controller is expected to see this signal high every time before the start of the write transfer. This input should be synchronized to the SCL clock domain.



**Note** CRC data is not pushed out of the Controller and the CRC error is calculated internal to the controller.

Figure 2-15 shows the timing diagram for Private Write Transfer in HDR-DDR Mode.

**Figure 2-15 Timing Diagram for Private Write Transfer in HDR-DDR Mode**



## 2.8.4 Private Read Transfer with Eight Bit Interface

### 2.8.4.1 Data Transfer with Sub-address

The following signals represent Private Read Transfer with Sub-address:

- **Wakeup:** The wakeup signal indicates to the application that some transfer request from the Master is meant for this slave and it must be ready to receive or send the data for the request.  
It is asserted when the slave address matches with the incoming address on the line after START, and de-asserted when after START there is no address match or when a STOP is detected on the I3C bus.
- **Sub\_addr:** Master can read the slave data from a particular sub-address. To specify the sub-address from which the Master wants to start reading the slave data, the Master first performs a write transfer indicating the sub-address. The size of the sub-address is already known to slave application. The slave application is expected to fetch the data from the received sub-address and assert the read data valid in stipulated time. The master then performs read transfer after Re-Start, to start reading slave data from specified sub-address. The sub-address is incremented once the controller accepts the current data presented by the application.
- **Slv\_lite\_rd\_ongoing:** This signal is asserted whenever there is an address match for the Master Read transfer targeted for this slave. This signal remains asserted until the read transfer is terminated by the Slave-Lite or the I3C Master has done an early termination.

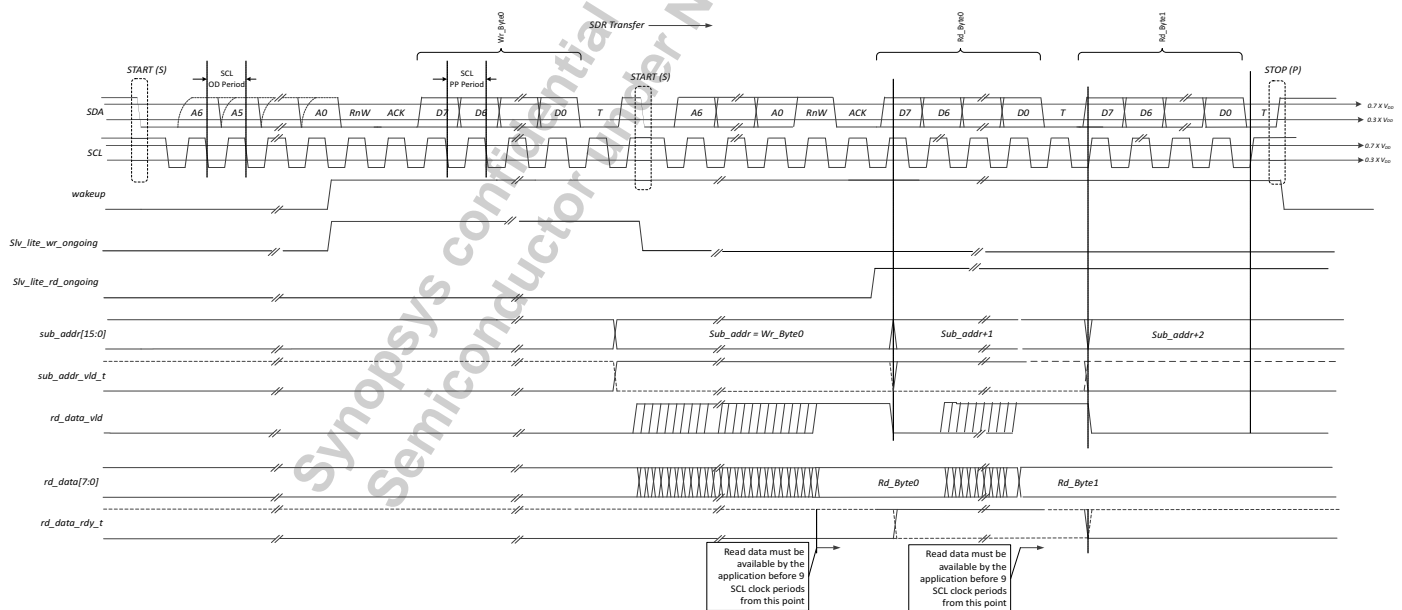
- **Sub\_addr\_vld\_t**: The sub\_addr\_vld\_t is toggled when a new sub\_addr is presented to slave application. The application has to fetch data from the new sub-addr and present to the controller in stipulated time. This signal toggles if a sub-address is successfully received. During the read transfer, this signal toggles one clock after ACK phase and T-bit phase (that is, after the controller accepts the data). This signal is not toggled if parity error is detected during the reception of the sub-address. In such cases, the sub\_addr is not updated and retains the value.
- **rd\_data\_rdy\_t**: This signal toggles when the controller accepts the current data presented by the application. The application has to fetch and provide next data (from new sub\_addr) within the stipulated time interval:
  - **SDR mode**: The rd\_data should be valid withing 8 SCL clocks after rd\_data\_rdy\_t. If the SCL frequency is 12.5 MHz, the approximate time for 8 SCL clocks is 640 ns

The rd\_data\_rdy\_t signal also toggles if a slave read request is NACK'ed because of read data not valid. This is to inform the application that a read request has come and is NACK'ed because of unavailability of data.

- **rd\_data\_vld, rd\_data**: Once the sub\_addr\_vld\_t toggles, the slave application has to present the new data to the controller within stipulated time. If the data is not valid by the time when the read request comes, the slave address is NACK'ed. rd\_data\_vld is asserted when valid data corresponding to the sub-address is ready to be sampled by the controller. Once asserted rd\_data\_vld should not be de-asserted until rd\_data\_rdy\_t toggles. The next data should be valid in stipulated time. If rd\_data\_vld is not asserted when rd\_data\_rdy\_t toggles, the read transfer is terminated by the slave after sending the last valid data presented to the controller.

Figure 2-16 shows the timing diagram for Private Read Transfer with Sub-address.

**Figure 2-16 Timing Diagram for Private Read Transfer with Sub-address**



#### 2.8.4.2 Data Transfer without Sub-address

This mode can be selected by driving zero on the input port sub\_addr\_size.

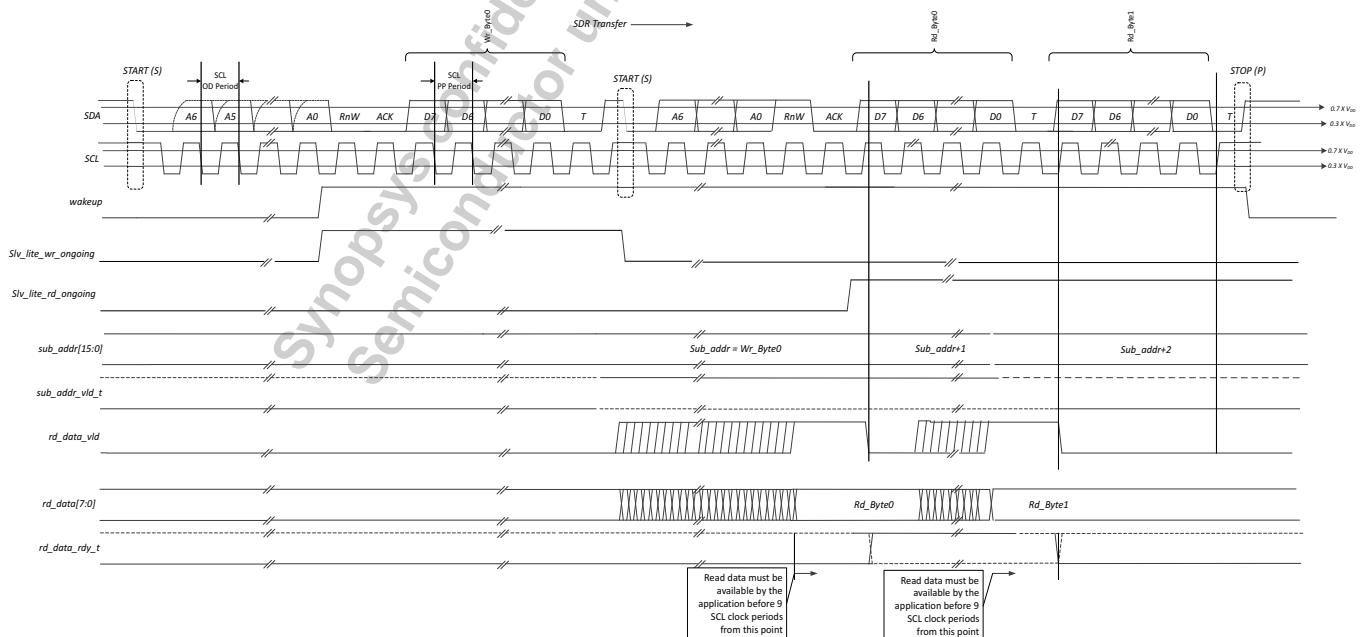
The following signals represent Private Read Transfer without Sub-address:

- **Wakeup:** The wakeup signal indicates to the application that some transfer request from the Master is meant for this Slave and it must be ready to receive or send the data for the request. It is asserted when the slave address matches with the incoming address on the line after START, and de-asserted when there is no address match after START or when a STOP is detected on the I3C bus.
- **Slv\_lite\_rd\_ongoing:** This signal is asserted whenever there is an address match for the Master Read transfer targeted for this slave. This signal remains asserted until the read transfer is terminated by the Slave-Lite or the I3C Master has done an early termination.
- **Sub\_addr:** This signal is Don't Care in this mode. There is no need to send the sub-address as write data. The read data presented by the application is sent to Master on read request.
- **Sub\_addr\_vld\_t:** This signal never toggles in sub\_addr\_size=0 mode.
- **rd\_data\_rdy\_t:** This signal toggles when controller accepts the current data presented by the application. The application has to fetch and provide next data (from new sub\_addr) within the stipulated time interval depending on the mode of operation:
  - **SDR mode:** The rd\_data should be valid within 8 SCL clocks after rd\_data\_rdy\_t

The rd\_data\_rdy\_t signal also toggles if a slave read request is NACK'ed because of read data not valid. This is to inform the application that a read request has come and is NACK'ed because of unavailability of data.
- **rd\_data\_vld, rd-data:** The slave application has to present the new data to the controller within stipulated time. If data is not valid by the time when read request comes, the slave address is NACK'ed. Rd\_data\_vld is asserted when valid data is ready to be sampled by the controller. Once asserted rd\_data\_vld should not be de-asserted until rd\_data\_rdy\_t toggles. The next data should be valid in stipulated time. If rd\_data\_vld is not asserted when rd\_data\_rdy\_t toggles, the read transfer is terminated by the slave after sending the last valid data presented to the controller.

Figure 2-17 shows the timing diagram for Private Read Transfer without Sub-address

**Figure 2-17 Timing Diagram for Private Read Transfer without Sub-address**





## 2.8.5 Private Read Transfers with 32 Bit Interface

### 2.8.5.1 SDR Mode with Byte Enable without Sub-address

In 32 bit interface, the Slave-Lite application must program the byte enable bits (`rd_data[33:32]`) to indicate the valid bytes in the 32 bit data. The byte enable is encoded as shown in the [Table 2-4](#).

**Table 2-4 Bytes Enabled**

Byte Enable ( <code>rd_data[33:32]</code> )	Bytes Valid
2'h0	1 byte
2'h1	2 bytes
2'h2	3 bytes
2'h3	4 bytes

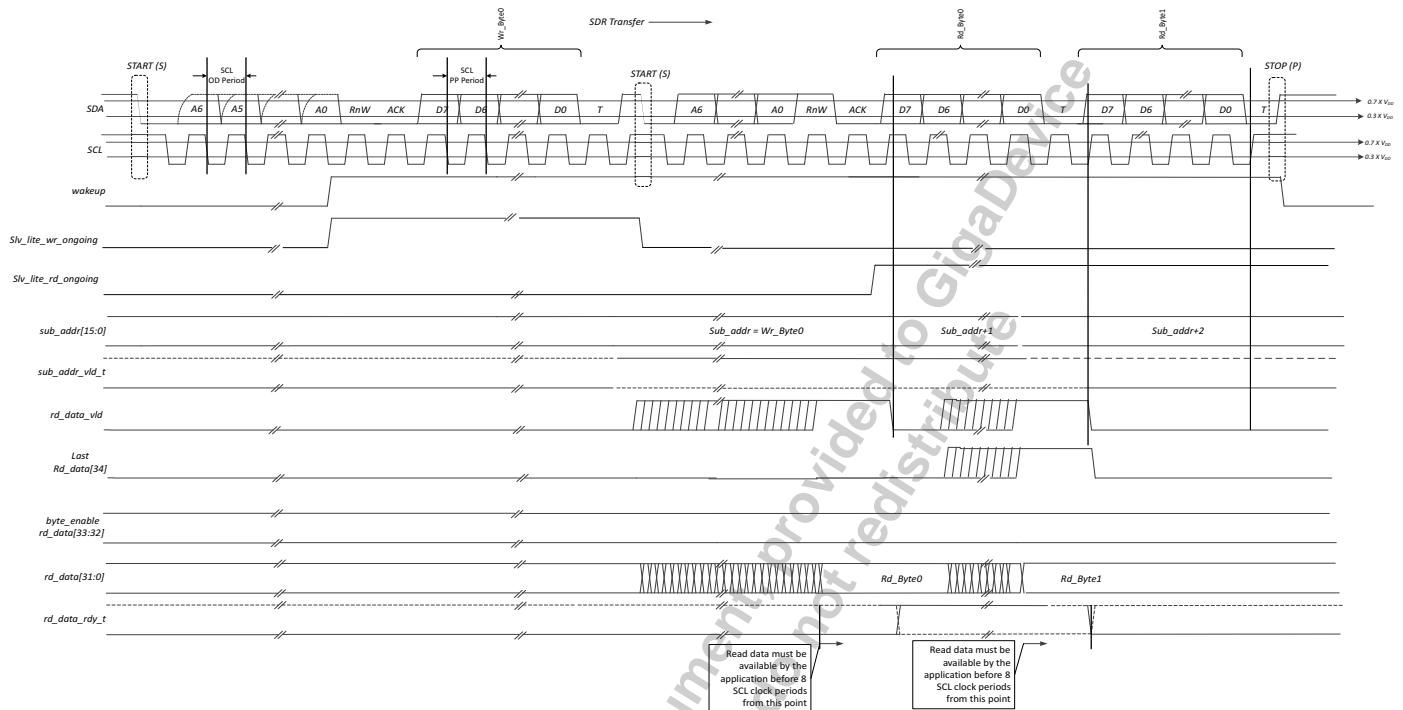
In SDR Mode all 32 bits of data can be valid depending on the requirement of the Slave-Lite application. The wider interface helps in receiving the `rd_data_rdy_t` at greater intervals such that the Slave-Lite application can be prepared with the data that is to be transferred.

The following signals represent Private Read Transfers in SDR mode:

- The last signal (`rd_data[34]`) is to be asserted for the last packet of data from the Slave-Lite application.
- Wakeup: The wakeup signal indicates to the application that some transfer request from the Master is meant for this slave and it must be ready to receive or send the data for the request. It is asserted after START the slave address matches with the incoming address on the line, and de-asserted after START there is no address match or when a STOP is detected on the I3C bus
- `Slv_lite_rd_ongoing`: This signal is asserted whenever there is an address match for the Master Read transfer targeted for this slave. This signal remains asserted until the read transfer is terminated by the Slave-Lite or the I3C Master has made an early termination.
- `rd_data_rdy_t`: This signal toggles when the controller accepts the current data presented by the application.

The slave application has to present the new data to the controller within stipulated time. If data is not valid by the time when read request comes, the slave address is NACK'ed. `Rd_data_vld` is asserted when valid data is ready to be sampled by the controller. Once asserted `rd_data_vld` should not be deserted until `rd_data_rdy_t` toggles. The next data should be valid in stipulated time. If `rd_data_vld` is not asserted when `rd_data_rdy_t` toggles, the read transfer is terminated by the slave after sending the last valid data presented to the controller.

[Figure 2-18](#) shows the timing diagram for Private Read Transfer in SDR Mode.

**Figure 2-18 Timing Diagram for Private Read Transfer in SDR Mode**

### 2.8.5.2 HDR Mode with Byte Enable with Sub-address

In HDR Mode of operation, the ready (`rd_data_rdy_t`) toggles approximately after 24 bits of data is sent. This gives the Slave-Lite application to pre-fetch the next data.

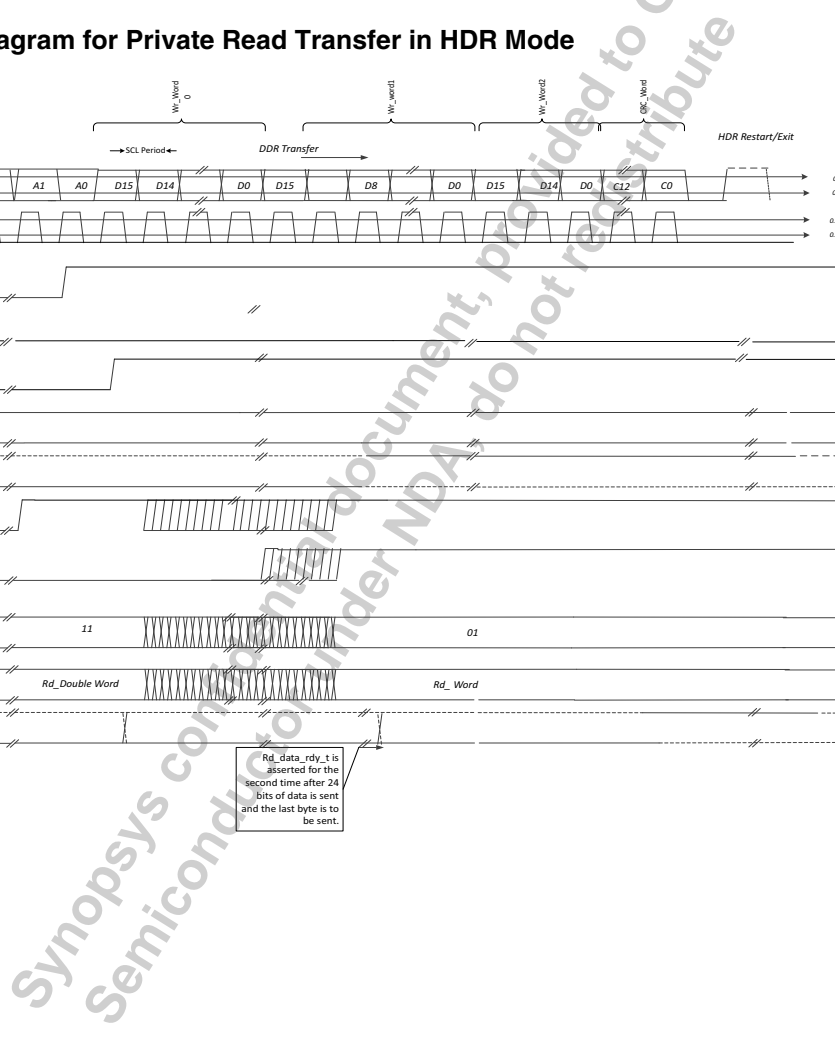
The following signals represent Private Read Transfers in HDR mode:

- **Wakeup:** The wakeup signal indicates to the application that some transfer request from the Master is meant for this slave and it must be ready to receive or send the data for the request. It is asserted when the slave address matches with the incoming address on the line, and de-asserted when there is no address match or when a STOP is detected on the I3C bus
- **Slv\_lite\_wr\_ongoing:** This signal is asserted whenever there is an address match for the Master Write transfer targeted for this slave. This signal remains asserted until the write transfer is over from the Master.
- **Slv\_lite\_rd\_ongoing:** This signal is asserted whenever there is an address match for the Master Read transfer targeted for this slave. This signal remains asserted until the read transfer is terminated by the Slave-Lite or the I3C Master has made an early termination.
- **Sub\_addr:** Master can read the slave data from a particular sub-address. To specify the sub-address from which the Master wants to start reading the slave data, the Master first performs a write transfer indicating the sub-address. The size of the sub-address is already known to slave application. The slave application is expected to fetch the data from the received sub-address and assert the read data valid in stipulated time. The Master then performs read transfer after Re-Start, to start reading slave data from specified sub-address. The sub-address is incremented once the controller accepts the current data presented by the application.
- **Sub\_addr\_vld\_t:** This signal toggles whenever the `rd_data` is internally latched into the controller and is ready to be sent on the I3C line.



- The Slave application must present the new data to the controller within stipulated time. If data is not valid by the time when read request comes, the slave address is NACK'ed. Rd\_data\_vld is asserted when valid data is ready to be sampled by the controller. Once asserted, rd\_data\_vld should not be deserted until rd\_data\_rdy\_t toggles. The next data should be valid in stipulated time. If rd\_data\_vld is not asserted when rd\_data\_rdy\_t toggles. The Slave terminates the read transfer after sending the last valid data presented to the controller.

### Figure 2-19 Timing Diagram for Private Read Transfer in HDR Mode



## 2.9 Support for PEC in Private Transfers for JESD403-1 Compliance

The JESD403-1 specification specifies packet error checking for both CCC and Private transfers. The PEC byte is appended at the end of the transfer, which is required to be decoded and checked for error. The Packet Error Checking is applicable only in I3C mode as specified in the specification. The PEC feature for JESD403-1 compliance is not supported in the current release. Eventually, DWC\_mipi\_i3c support for JESD403-1 for Slave-Lite configuration is limited to two configuration that are representative of the JEDEC use case and can be found under “Validated Configurations” section of DWC\_mipi\_i3c Controller User Guide.

The PEC is a CRC-8 value calculated on all the messages bytes except for START, STOP, REPEATED START conditions or T-bits, ACK and NACK and IBI header (7'h7E followed by W=0) bits. The 8-bit PEC is appended at the end of all transactions if PEC is enabled. The device (either in master or slave mode) appends the PEC as the last byte of transaction while transmitting and checks for the correctness of PEC received as the last byte of transaction while receiving.



### Note

If PEC error is detected, the slave NACK's all transfers until Stop (P) is detected.

### 2.9.1 PEC Generator Algorithm

The polynomial used for CRC-8 calculation is

$$C(X) = X^8 + X^2 + X + 1$$

The CRC8 calculation for one bit at a time is as follows. Since PEC is transmitted only in SDR mode, one bit can be sampled at every posedge of SCL clock. The CRC is computed for every bit which is sampled or driven on the line.

```
next_CRC8[0] = SDA_r[0] ^ CRC8[7];
next_CRC8[1] = SDA_r[0] ^ CRC8[0] ^ CRC8[7];
next_CRC8[2] = SDA_r[0] ^ CRC8[1] ^ CRC8[7];
next_CRC8[3] = CRC8[2];
next_CRC8[4] = CRC8[3];
next_CRC8[5] = CRC8[4];
next_CRC8[6] = CRC8[5];
next_CRC8[7] = CRC8[6];
```

### 2.9.2 Private Read/Write with PEC Enabled

The Master appends PEC byte at the end of the private write transfer. The PEC byte is computed on both slave address and private data. The slave flags a PEC error if there is any mismatch between internally computed PEC and the PEC byte received from Master.

The Slave appends PEC byte at the end of private read transfer. The PEC byte is computed on both slave address and private read data. The Master must check whether the internally computed PEC is same as the PEC received from Slave.

**Figure 2-20 Private Read/Write with PEC Enabled**

START (S/Sr)	Slave Address (SA, w/r)	Data Payload (DB, T)	PEC Byte (CRC Byte,T)	STOP (P/Sr)
-----------------	----------------------------	-------------------------	--------------------------	----------------

### 2.9.3 PEC calculation and Check in Slave Mode

The input signal 'PEC enable' is driven by the application as PEC can be enabled or disabled through DEVCTRL CCC or by modifying JESD403-1 application-specific MR18 register. The application-specific MR18 register is not maintained internally by the controller. The application has to maintain the application-specific MR18 register and controller gives sufficient signals for application to determine whether Master intends to modify the MR18 register. The application has to set the PEC enable port if PEC is enabled by Master and has to de-assert it when PEC is disabled by the Master.

In Slave mode, controller must check for PEC error during Master write transfers and must calculate and append PEC byte while transmitting read data. PEC is calculated on all address, CCC and data bits except 7E/W as per the JESD403-1 Specification. The PEC is reset to initial value on Start, Restart, or Stop.

#### 2.9.3.1 PEC calculation for Write Transfers

During Master private/CCC write transfer, the Slave controller calculates PEC on all relevant bytes received from the Master. The Slave computes the PEC for each bit and updates the 'slv\_pec\_error' output at the end of each byte. At the end of each byte, received byte is compared with the internally calculated CRC8 value. The output 'slv\_pec\_error' is set if the received byte is not same as the internally calculated CRC8 value. The last byte of the transaction is PEC byte and PEC byte, if correct, matches with internally calculated CRC8 value making slv\_pec\_error low. Hence, in a proper write transfer, 'slv\_pec\_error' is high until the penultimate byte is de-asserted at the end of the last byte. This is required as the Slave does not have any prior information that it is the last byte. Only after the transfer is complete, the Slave knows that it is the last byte. Hence, 'slv\_pec\_error' is updated at the end of every byte.

The qualifier for slv\_pec\_error are de-assertion of 'slv\_lite\_wr\_ongoing' and 'ccc\_ongoing' (see "[Slave Interface Signals](#)" on page 103). During CCC write transfers, de-assertion of ccc\_ongoing indicates CCC transfer is complete and if slv\_pec\_error is set, it indicates that the CCC transfer has PEC error. During private write transfers, de-assertion of slv\_lite\_wr\_ongoing' indicates write transfer has completed and if slv\_pec\_error is set, it indicates that the private write transfer has PEC error. So slv\_pec\_error alone going high does not mean PEC error. During de-assertion of 'ccc\_ongoing' or 'slv\_lite\_wr\_ongoing', if slv\_pec\_error is set it indicates the last transfer has PEC error.



#### Note

wr\_data\_vld\_t toggles for PEC byte too. The slave application should always ignore the last byte of write transfers if PEC is enabled.

#### 2.9.3.2 PEC Calculation for Read Transfers

The Slave should calculate PEC for private and directed CCC (only for CCCs specified in JESD403-1 specification) read transfers. PEC can be calculated by providing the transmit data bit by bit to the PEC generator. The Slave should append the PEC data at the end of every transfer if PEC is enabled (slv\_pec\_enable-1). No intervention is required from application to send PEC. The Slave controller

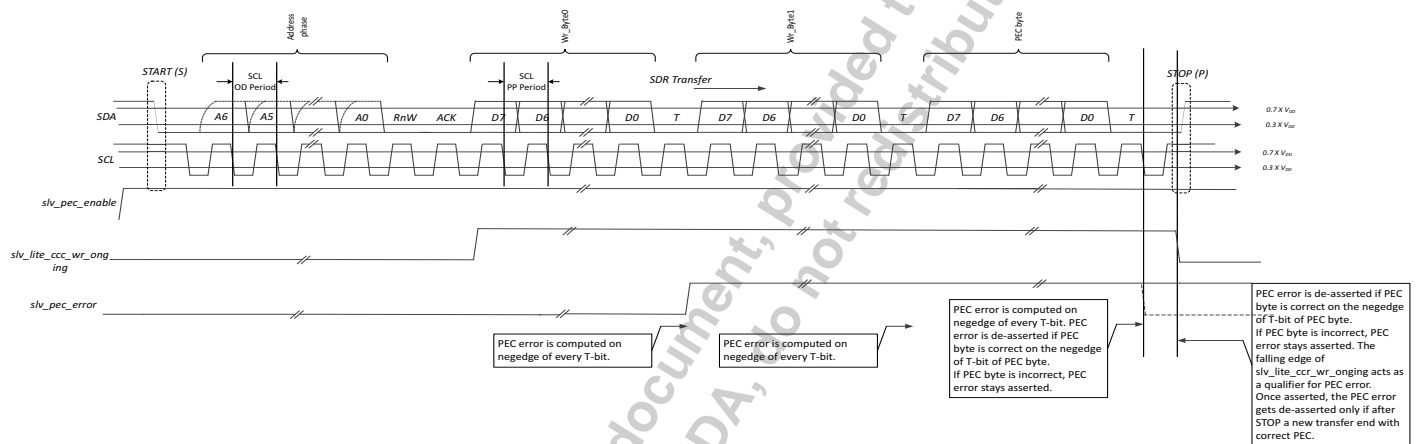
automatically computes the CRC8 and appends the PEC byte at the end of transfer. Whenever controller samples new `rd_data`, `rd_data_rdy_t` toggles and new data is sent on I3C line. Since PEC byte is internally generated, `rd_data_rdy_t` does not toggle when controller is sending PEC byte.

**Note**

`rd_data_rdy_t` does not toggle when controller is sending PEC byte.

## 2.9.4 Timing Diagram of Private Write with PEC Error

Figure 2-21 Private Write with PEC Error



## 2.10 Error Handling

JESD403-1 specification requires some errors mentioned in I3C specification to be handled differently. Some error detection and recovery are required by JESD403-1 application while some others are not required. This section elaborates the hooks provided by the Slave controller in Slave-Lite mode of operation to control error handling.

### 2.10.1 Connections of Inputs Required for Error Handling

The following inputs/hooks are provided in the controller for error handling for JESD403-1 compliance and I3C mode of operations.

The connections of these inputs for JESD403-1 compliance and I3C mode of Operations are as follows:

**Table 2-5 Connections of Inputs for Error Handling**

Input Name	No. of Bits	JESD403-1 Mode Connection	I3C Mode Connection	Comments
slv_s0_disable	1	Should be connected to logic 1	Should be connected to logic 0	For JESD403-1 compliant mode of operation, S0 error must be disabled as JESD403-1 use-case supports the usage of the I3C reserved header 7e and its derivatives.
slv_s1_err_rst	1	Should be connected to logic 1	Should be connected to logic 1.	This input is required to recover from S1 error after both SCL and SDA is high for 60usec. This counter calculating 60usec should be in the application as Slave-Lite uses SCL as a clock which is intermittent. This functionality is not verified and the input pin should be connected to logic 1
slv_perr_ack_onc	1	Should be connected to logic 1	Should be connected to logic 0	This input when connected to logic 1 affects the controller as follows: Recover from any parity, pec, S1, S2 error on STOP. Does not disable the detection of START/STOP detection and does not wait for exit pattern/60usec recovery during S1 error.
slv_par_err_dis	1	Should be connected to a JESD403-1 application register (MR register) holding parity error enable/disable status	Should be connected to logic 0	For JESD403-1 compliance, parity error is enabled or disabled based upon DEVSTR_L CCC. The slv_par_err_dis, if asserted by the application, disables detection of parity error. In I3C mode, the I3C protocol supports detection of parity error hence the input should be connected to logic 0.

Input Name	No. of Bits	JESD403-1 Mode Connection	I3C Mode Connection	Comments
slv_par_err_ext	1	Should be connected to a JESD403-1 application register (MR register) holding parity error status	Should be connected to logic 0	For JESD403-1 compliance mode, the status of parity error should not be cleared on GETSTATUS CCC. This input, if set, keeps the parity error status set even after GETSTATUS CCC.
slv_pec_err_ext	1	Should be connected to a JESD403-1 application register (MR register) holding PEC error status	Not Applicable. PEC should not be configured in I3C mode of operation. In I3C configuration when PEC support is not selected, this input should not be present in RTL.	For JESD403-1 compliance mode, the status of PEC error should not be cleared on GETSTATUS CCC. This input, if set, keeps the PEC error status set even after GETSTATUS CCC.
slv_pec_enable	1	Should be connected to a JESD403-1 application register (MR register) holding pec error enable/disable status	Not Applicable. PEC should not be configured in I3C mode of operation. In I3C configuration, when PEC support is not selected this input should not be present in RTL.	

**Note**

The IP features/interface for JESD403-1 compliance is not supported in the current release. The inputs to the IP listed in [Table 2-5](#) are expected to follow the recommendations given under column I3C Mode Connection.

Eventually, DWC\_mipi\_i3c support for JESD403-1 for Slave-Lite configuration will be limited to two configuration that are representative of the JEDEC use case and can be found under “Validated Configurations” section of DWC\_mipi\_i3c Controller User Guide.

## 2.10.2 Disabling of S0 Error Detection for JESD403-1 Compliance

S0 error is detected whenever the 7'h7E header is received incorrectly by the Slave controller. The derivative of 7'h7E like 7'h3E, 7'h5E, and so on makes the controller detect an S0 error and halt all its operation by issuing a NACK to the incoming transfers from I3C master until HDR exit pattern is detected.

The S0 error should be disabled if the application (like JESD403-1) requires the derivatives of 7'h7E as valid address for the slave.

The application can disable the error detection for S0 by asserting the input pin “slv\_s0\_disable”.

If the input pin slv\_s0\_disable is asserted, the controller does not check for S0 error and does not enter halt state and waits for exit pattern detection. The input slv\_S0\_disable must be synchronized with respect to SCL clock.

### 2.10.3 S1 Error Detection and Recovery

The S1 error is detected when a CCC type with Parity error is received. Once the S1 error is detected by the controller, it goes to a halt state and NACK's all the transfer coming from the I3C master until the control recovers from the error state.



#### Note

The recovery mechanism for other errors like S3-S5 happens after a STRAT, Re-START, or STOP as given in Mipi I3C specification, and does not require special handling. For JESD403-1 application, the S3-S5 errors are not supported and the controller does not expect these errors to happen on the I3C line.

#### 2.10.3.1 Recovery in I3C Mode of Operation

In I3C mode of operation, on detection of S1 error, the detection logic for START/STOP is disabled. The `wr_par_err_t` signal toggles indicating to the application a parity error has been detected.

The controller can recover only by the Exit Pattern Detection method give as follows:

- Recovery through Exit Pattern Detection: The controller resumes accepting all transfers after the HDR exit pattern is detected. Until the exit pattern is detected, the controller keeps NACK'ing the transfers from the I3C master.

#### 2.10.3.2 S1 Error Recovery in JESD403-1 Compliance Mode of Operation

In JESD403-1 mode of operation, the S1 error recovery based on HDR Exit Pattern is not possible as JESD403-1 applications do not support it. On detection of S1 error, the internal parity error flag is set and the output signal `wr_par_err_t` toggles indicating a parity error is detected after CCC type is received. The controller NACK's all transfers after the detection of parity error until STOP.

As required by the JESD403-1 specification, the parity error is not cleared on GETSTATUS.

### 2.10.4 Parity Error Recovery

The Slave controller decodes the parity error internally provided the input `slv_par_disable` is de-asserted. The JESD403-1 application can enable or disable the parity error based on DEVCTRL CCC. Once a parity error is decoded, the Slave controller has the following two modes of recovery depending on mode of operation:

- Mode 1 I3C: The Slave controller NACKs all transfers after the parity error has occurred until START or STOP (Refer "[Parity Error Handling in I3C Mode](#)" on page 65). The parity error bit is latched in internal status register and the content transmitted to the Master in response to GETSTATUS CCC. The internal status register gets flushed on receiving GETSTATUS CCC. The controller does not hold the parity error status after GETSTATUS CCC is received.
- Mode 2 JESD403-1: The Slave controller ACK's the next transfer only after STOP as required by JESD403-1 specification. The detected parity error is latched to an internal status register and is transmitted to the Master along with the status of the external parity error input pin as given in section "[Parity Error Recovery in JESD403-1 Compliance Mode of Operation](#)" on page 64.

To enable the second mode of operation, the input pin `slv_par_err_ack_onc` (slave parity error ACK on completion) must be asserted. This input pin must be tied to 1 (JESD403-1 compliance mode) or 0



(I3C Mode) depending on the mode of operation required and its driving value cannot be changed during run time.

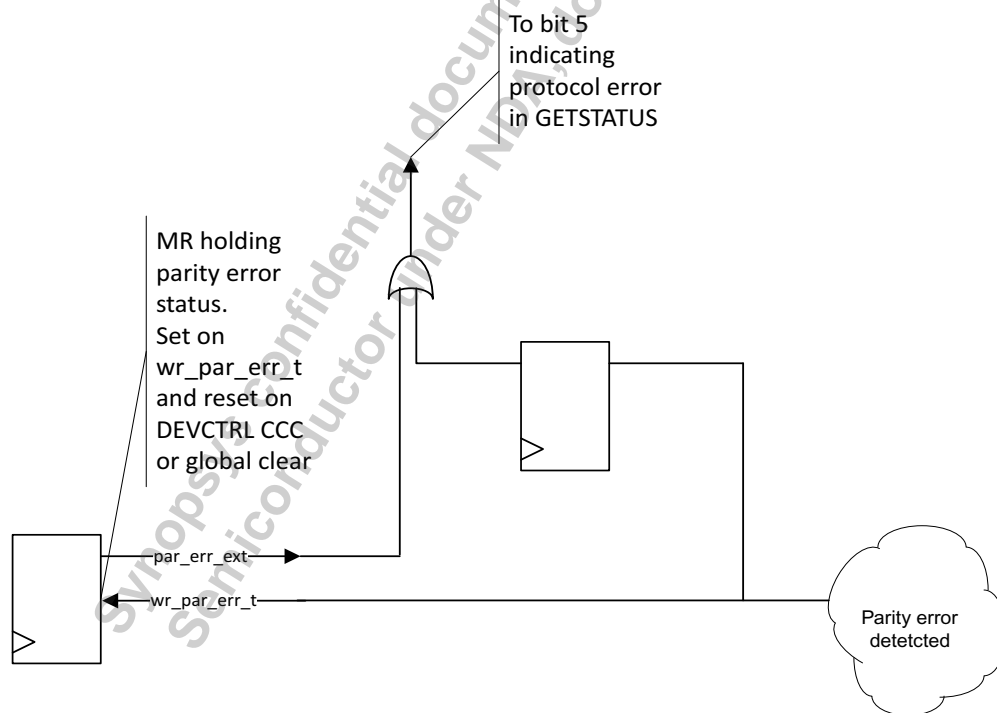
#### 2.10.4.1 Parity Error Recovery in JESD403-1 Compliance Mode of Operation

The protocol error bit [5] of GETSTATUS is driven by the following two sources:

- The internal status register, which gets asserted on a parity error detection. This internal status register gets cleared once a GETSTATUS CCC is received from the I3C Master.
- The input pin `par_err_ext` driven by the application.

The parity error indicated by the toggle of the signal `wr_par_err_t` should set the JESD403-1 application specific register (MR register) meant for holding the status of parity error. The input “`par_err_ext`” should be driven by the same JESD403-1 application specific register (MR register) holding the parity status. The parity error status stored in the external register (MR register) in the application continues to drive the protocol error bit[5] of GETSTATUS until it is cleared by global clear command using DEVCTRL CCC. Thus, in JESD403-1 compliance mode, the controller holds the parity error status even after GETSTATUS is received through the input “`par_err_ext`”.

**Figure 2-22 JESD403-1 Compliance Mode Parity Error Scheme**



The “`par_err_ext`” input along with the internal parity error flag drives the parity error status when a GETSTATUS CCC is requested by the I3C Master.



**Note**

For PEC error Mode, 2 is always applicable. A separate input port `slv_pec_err_ext` is provided to consider the external PEC error latched by the application. The input `slv_pec_err_ext` must be connected to the output of the register in the application holding the PEC error.

#### 2.10.4.2 Parity Error Handling in I3C Mode

Parity error during Private transfer from Master

- The `DWC_mipi_i3c` in Slave-Lite mode of operation, sets the protocol error flag which can be read by the Master on issuing GETSTATUS CCC where bit 5 of the CCC is set. The occurrence of parity error is indicated to the application by the `wr_par_err_t` toggle signal. The controller ACKs all transfers after the transfer having parity error. To NACK any private transfers application can de-assert the `wr_data_rdy` and the `rd_data_vld` inputs.
- The bit 5 of GETSTATUS CCC is also set for S1 and S2 errors specific to CCCs. The recovery mechanism for S1 and S2 error is as given in I3C specification, that is on exit pattern and on START or STOP conditions respectively.

#### 2.10.5 S2 Error Recovery

The S2 error occurs when there is a parity error detected on the payload packet of a CCC. The recovery form this error is different for JESD403-1 compliance mode and I3C mode of operation. In JESD403-1 compliance mode of operation, the input pin “`slv_per_ack_onc`” should be connected to 1 to make the controller come out from S2 error on STOP.

##### 2.10.5.1 Recovery in I3C Mode of Operation

In I3C mode of operation, if a S2 error is detected, the CCC is ignored if the CCC is used internal to the controller. The `wr_par_err_t` toggles indicating the occurrence of parity error.

If the CCC is not used internally, it is passed to the CCC channel. The `ccc_vld_t` and all associate CCC channel signals (`ccc_ongoing`, `ccc_wr_data`, and so on) toggles with `wr_par_err_t`. The application should discard the CCC data and take no action for the received CCC. The protocol error bit for GETSTATUS reflects the parity error status. The parity error status is cleared after GETSTATUS CCC. Refer to “[Parity Error Handling in I3C Mode](#)” on page 65. The controller recovers from S2 error on the first START or STOP received on the line after the detection of error.

##### 2.10.5.2 Recovery in JESD403-1 Compliance Mode of Operation

The behavior in JESD403-1 compliance mode of operation is identical to that of I3C mode of operation, except the recovery for S2 error happens only on STOP and not on START. The GETSTATUS protocol error bit 5 is driven from two sources as explained in section “[Parity Error Recovery in JESD403-1 Compliance Mode of Operation](#)” on page 64. Thus, in JESD403-1 compliance mode, the controller holds the parity error status even after GETSTATUS is received through the input “`par_err_ext`”.

## 2.10.6 Asserting slv\_par\_err\_dis and slv\_pec\_enable for JESD403-1 Compliance

By default, in the DWC\_mipi\_i3c controller, parity error checking is enabled and packet error checking is disabled. The Master can enable or disable these functions with DEVCTRL CCC. The controller does not decode the DEVCTRL CCC from Master, the application can enable or disable the parity and packet error check functions using the input ports slv\_par\_err\_dis and slv\_pec\_enable. After receiving the DEVCTRL CCC, there can be a delay from the application in asserting/de-asserting of slv\_par\_err\_dis and slv\_pec\_enable signals. The DWC\_mipi\_i3c controller allows a delay of 9 SCL clocks in asserting or de-asserting slv\_par\_err\_dis and slv\_pec\_enable, after the end (Stop) of DEVCTRL CCC. This is to ensure that application enables/disables the parity and packet error check functions within the address phase of next transfer after DEVCTRL CCC. If slv\_par\_err\_dis or slv\_pec\_enable are asserted/de-asserted during the data phase of transfer, it may result in incorrect reporting of parity and PEC error. Hence, the application should assert/de-assert the slv\_par\_err\_dis and slv\_pec\_enable input signals within 9 SCL clocks after the end of DEVCTRL CCC.

Synopsys confidential document, provided to SolvNet  
Semiconductor under NDA, do not redistribute

## 2.11 I3C vs I2C Role Selection

The DWC\_mipi\_i3c Slave-Lite can be selected as an I3C slave or as an I2C Slave by the following two ways:

1. The input pin “mode\_i2c”, if tied to 1, brings up the Slave Controller as an I2C slave. Once selected as an I2C slave, the role cannot be changed until reset and after the input pin “mode\_i2c” is made zero.
2. Adaptive Selection: The adaptive selection is possible only when the input pin “mode\_i2c” is tied to 0. The DWC\_mipi\_I3C controller is in I2C mode during initial power up as it does not have any Dynamic Address (DA) assigned to it. The mode changes to I3C only when a DA is assigned by any of Dynamic Address assignment procedure given in MIPI I3C Specification. The mode again changes back to I2C when RSTDAA CCC is issued or on hardware reset.

### 2.11.1 Conditions for Mode Change

On Power-up, the Slave controller is in I2C mode and it changes state under the following conditions:

- The Slave controller changes to I3C mode when a dynamic address is assigned by any of the Dynamic Address assignment procedures given in I3C specification.
- The Slave controller changes to I2C mode when a RSTDAA CCC is received from the Master. The Slave controller responds to transfers matching its static address in I2C mode.
- On Bus reset or Power On, reset the controller is in I2C mode.

### 2.11.2 Conditions for Glitch Filter Enable/Disable

The glitch filter is disabled by the controller when the first 7E header is received. Once disabled, it is in disable state until Power On Reset.

### 2.11.3 Conditions for Wakeup

The DWC\_mipi\_i3c controller in Slave-Lite mode of operation supports low power mode usage with the output signal “wakeup”. This signal, when asserted, can be used by the application to come out of power saving mode or clock gating mode.

Wakeup signal is asserted in the following conditions:

**Table 2-6 Conditions for Wakeup**

Assertion of Wakeup	De-assertion of Wakeup	Description
CCC Broadcast		

Assertion of Wakeup	De-assertion of Wakeup	Description
Wakeup is asserted when the first bit of CCC is received after 7E.	End of CCC transfer as per I3C specification on STOP or Re-Start and 7E header	<p>The broadcast CCC can be consumed internally by the controller, or it can be passed to the application through the CCC channel. The application can wakeup when this signal is asserted. The minimum time to response when wakeup is asserted is nine SCL clocks. The application after wakeup assertion can monitor the <code>ccc_vld_t</code> signal to understand broadcast CCC is consumed internal to the controller or passed to the Slave-Lite application.</p> <p>For ENTDAACCC, the wakeup does not get de-asserted on a parity error received on Dynamic address received from Master and remains asserted until stop.</p> <p>For enthdr ccc, the wakeup behavior is as follows:</p> <ol style="list-style-type: none"> <li>1. The wakeup is asserted when broadcast enthdr ccc is received.</li> <li>2. It is de-asserted when the ccc byte is completed (after the ccc type is decoded).</li> <li>3. It is asserted again if there is an address match in DDR mode of operation.</li> <li>4. It is de-asserted again when there is address mismatch in DDR mode of operation.</li> </ol> <p>For RSTDAACCC, the wakeup does not get de-asserted in a restart but on an I2C address mismatch.</p>
CCC Directed		
Wakeup is asserted when the directed CCC is passed on to the application through the CCC channel. It is asserted along with <code>add_mtch_dir_t</code> .	End of CCC transfer as per I3C specification on STOP or Re-Start and 7E header	The wakeup is not asserted for directed CCCs consumed internal to controller. It is asserted only for CCCs passed to the application.
Private Transfers		
Wakeup is asserted when an address match occurs indicating the Master is trying to access this slave	The wakeup is de-asserted after an address mismatch or on STOP. It is also de-asserted after a re-start followed by 7E	The minimum time to response for the application when wakeup is asserted is nine SCL clocks. The application has to buffer the read and the write data so that the transfer can be ACK'ed.

**Note**

For any errors associated with CCC, the wakeup is de-asserted on detection of the error.

## 2.12 CCC Transfers with DWC MIPI I3C Slave-Lite Controller

### 2.12.1 Overview of CCC Transfers with DWC MIPI I3C Slave-Lite Controller

The Slave-Lite configuration collects the required CCC information either through configurable parameters or from input port signals, to enable the controller to handle the CCC transfers.

During run time, the controller does not involve the application to handle any CCC transfers (with the exception of Vendor specific CCC commands or unsupported CCCs). Upon receiving a CCC command from the I3C Master which is internally consumed by the controller, the controller handles it in one of the three ways – through configurable parameters, through port signals or through internal registers.

In I2C mode, the Master can send a broadcast CCC in open drain SDR mode starting with a 7e header. The Slave on detecting the 7e header identifies that the transfer from the Master as a CCC transfer. The slave decodes the CCCs in I2C mode and performs the functionality required by the issued CCC. The Slave controller operates in open drain SDR mode for the ongoing CCC transfer and abides with all the requirement (half cycle ACK for write CCC, 9th bit as parity bit for write) of the I3C protocol in open drain SDR mode for the ongoing CCC transfer. After completing CCC, the Slave controller falls back to I2C mode of operation for any private read or write transfers matching its static address. The requirement of accepting CCC only in I3C mode is removed.

The Slave can only accept broadcast write CCCs from the Master in I2C mode as directed CCC needs to have a Dynamic Address associated with the slave and hence not possible. The slave controller NACKs all CCC reads (Directed GET CCC) in I2C mode.

### 2.12.2 Description of CCC Transfers with Slave-Lite

#### 2.12.2.1 CCCs Handled through Configuration Parameters

Some CCC related parameters must be set while configuring the controller. These parameters are used to handle the read CCC transfers (listed in [Table 2-7](#)) from I3C Master. These parameters specify the capabilities and features supported by the controller. Hence, once configured, these cannot be changed.

**Table 2-7 CCS Handled through Configuration Parameters**

CCC	Related Parameters
GETBCR	IC_SLV_HDR, IC_SLV_BRIDGE, IC_SLV_OFFLINE_CAP, IC_SLV_IBI, IC_SLV_IBI_DATA, IC_SLV_DATA_SPEED_LIMIT
GETMXDS	IC_SLV_MXDS_MAX_WR_SPEED, IC_SLV_MXDS_MAX_RD_SPEED, IC_SLV_MXDS_CLK_DATA_TURN, IC_SLV_MXDS_MAX_RD_TURN
GETHDRCAP	IC_SLV_HDR_DDR, IC_SLV_HDR_TSP, IC_SLV_HDR_TSL

#### 2.12.2.2 CCCs Handled through Port Signals

The CCCs that are related to dynamic functionality of the controller are handled through port signals.

Unlike CCCs handled through configuration parameters, these CCCs can be changed during runtime. The data associated with write CCC is made available to the application through output ports and data associated with read CCCs is collected through input ports.

Table 2-8 lists the CCCs handled through port signals.

**Table 2-8 CCCs Handled through Port Signals**

CCC	Related Port Signal
ENTAS0	act_state
ENTAS1	act_state
ENTAS2	act_state
ENTAS3	act_state
SETMWL	mwL (At present the IP does not limit the data length based on MWL)
SETMRL	mrl (At present the IP does not limit the data length based on MRL)
GETSTATUS	act_mode, pending int
ENTDAA	All input ports and parameters related to GETPID, GETBCR and GETDCR
GETPID	slv_pid
GETDCR	slv_dcr

### 2.12.3 CCCs Handled through Internal Registers

Some CCCs are handled internally to the controller with no or minimal inputs from the user.

Table 2-9 lists the CCCs that are handled through internal registers.

**Table 2-9 CCCs Handled through Internal Registers**

CCC	Related Internal Register
ENEC	Updates the internal register and automatically controls the MR, SIR, and HJ requests from the controller.
DISEC	Updates the internal register and automatically controls the MR, SIR, and HJ requests from the controller.
RSTDAA	Clears internal dynamic address and address valid register.
ENTHDR0	Controller starts operating in HDR_DDR mode.
ENTHDR1	Controller starts operating in HDR_TSP mode.
ENTHDR2	Controller starts operating in HDR_TSL mode.
SETDASA	Static address and valid is provided through input ports. If static address matches with incoming address, internal dynamic address register is updated and valid is set after receiving SETDASA CCC.
SETNEWDA	Updates the internal dynamic address register
GETMWL	Transmits the value from the internal MWL register to the I3C Master

CCC	Related Internal Register
GETMRL	Transmits the value from the internal MRL register to the I3C Master

## 2.12.4 Handling of ENTDA, GETPID and GETDCR

It might not always be possible to determine the PID and DCR values at the time of RTL configuration. The configuration parameter `IC_SLV_UNIQUE_ID_PROG` provides these values at run time through straps as opposed to configuring the values during RTL generation.

When `IC_SLV_UNIQUE_ID_PROG` is set to 0

- Of the 48 bits of the Provisional ID, 44 bits are configured during RTL generation and only Instance ID (Bits[15:12]) is changeable at run time. Instance ID can be programmed using `inst_id` input port. The remaining 44 bits of PID value is obtained by the controller from the following Configuration Parameters:
  - `IC_SLV_MIPI_MFG_ID`
  - `IC_SLV_PROV_ID_SEL`
  - `IC_SLV_PART_ID`
  - `IC_SLV_PID_DCR`.
- The value of DCR (Device Characteristic Register) is also configured during RTL generation using the Configuration Parameter `IC_SLV_DCR_VALUE`.

When `IC_SLV_UNIQUE_ID_PROG` is set to 1

- This makes it possible for values of Provisional ID and DCR to be provided by driving the straps. The value of 48-bit Provisional ID can be provided by straps by driving strap input `slv_pid[47:0]`.

The value of DCR can be provided through strap by driving strap input `slv_dcr[7:0]`. Detailed description of the strap inputs provided for programming the PID and DCR values is as follows:

- PID
  - PID[47:33]: MIPI Manufacturer ID  
Sourced from `slv_pid[47:33]` input port
  - PID[32]: Provisional ID Type Selector  
Sourced from `slv_pid[32]` input port
  - PID[31:16]: Part ID  
Sourced from `slv_pid[31:16]` input port
  - PID[15:12]: Instance ID  
Sourced from `slv_pid[15:12]` input port
  - PID[11:0] Additional meaning  
Sourced from `slv_pid[11:0]` input port
- DCR  
Sourced from `slv_dcr` input port



## 2.13 Support for Vendor Specific CCC with Defining Bytes

The DWC\_mipi\_i3c controller in Slave-Lite mode of operation can handle any vendor specific CCCs either directed or broadcast. A separate channel for vendor specific CCCs is provided through which the Slave application can accept and provide vendor specific information.

The CCC channel for vendor specific CCC is also used to pass any CCC that are not supported by the controller. The detail explanation along with the table describing the supported and unsupported CCCs are explained in “[Handling of Unsupported CCC](#)” on page 82. The defining byte in CCCs helps in using the same CCCs to transmit or receive different commands from the I3C Master to the Slaves. The absence of a defining byte limits the usage of CCC to only one type of command for a CCC.

The Slave controller has the capability to decode the CCC and the associated defining byte and transfer it to the application for appropriate action.

Note that the defining byte support is only possible for vendor specific CCCs.

The vendor specific CCC can be a directed or a broadcast CCC. The CCCs might not be associated with defining bytes. The CCC channel is used to pass the received vendor specific CCC and the associated defining byte to the application.

The CCC type is placed in the 8-bit `ccc_type` interface. The associated defining byte, if present, is placed in `ccc_def_byte` interface. The toggle signal `ccc_vld_t` is used as a qualifier to sample the CCC type and the CCC defining byte information.

The Slave controller needs a configuration input to differentiate the defining byte only for a broadcast vendor specific CCC. Configure the broadcast CCC type which has a defining byte in `coreConsultant`. No input is required for directed CCC which demarcates the defining byte based on the re-start condition on the line. All vendor CCCs both broadcast and directed support the format with defining byte as well as the format without defining byte.

The Controller supports both formats of a vendor CCC which can have a defining byte and a format without the defining byte.

The vendor specific CCC may have a payload data associated with it. The payload data is transmitted to the application through the 8-bit output port `ccc_wr_data`. The toggle signal `ccc_wr_data_vld_t` is used as a qualifier to sample the payload data.



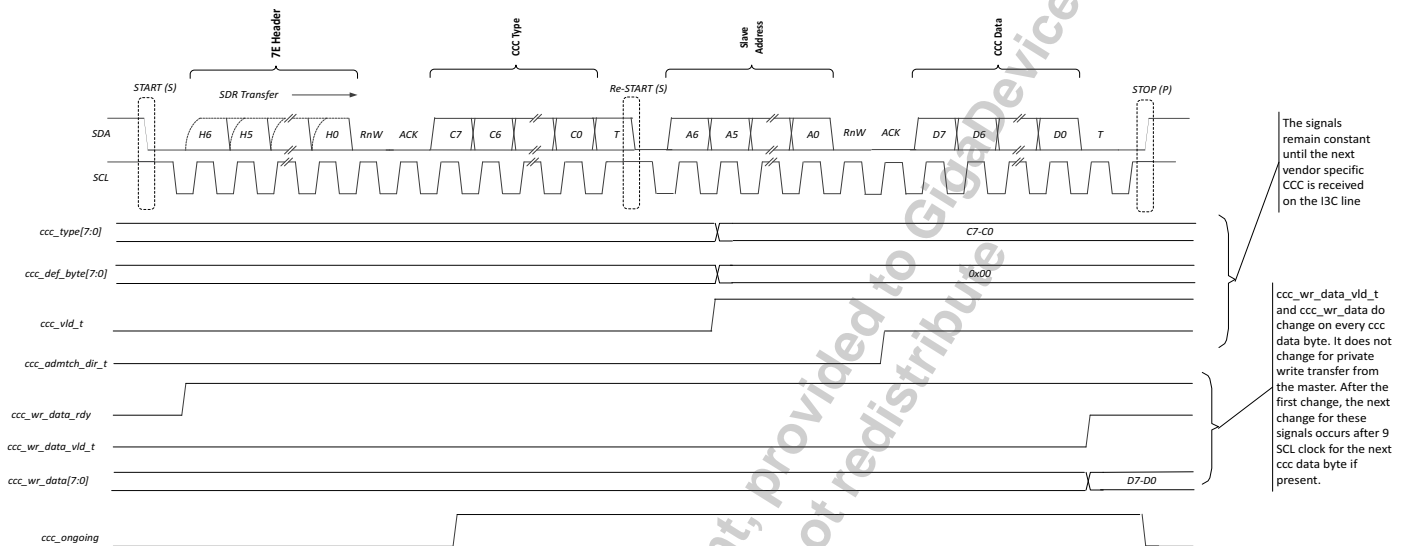
### Note

Parity error in the defining byte through a valid scenario is not supported as the recovery mechanism is not mentioned in the MIPI I3C specification.



### 2.13.1 Timing Diagram of Directed Write CCC without Defining Byte

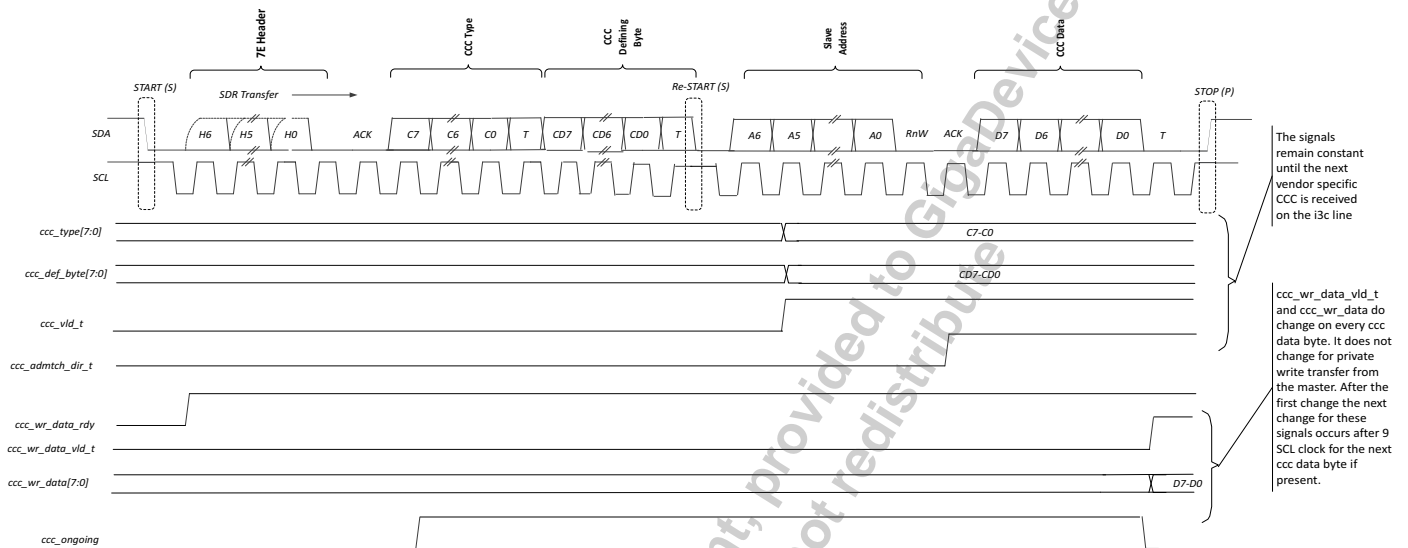
**Figure 2-23** Timing Diagram of Directed Write CCC without Defining byte



- When a ccc which is not consumed internal to the Slave is received, the ccc write channel is activated.
- The signals ccc\_type and ccc\_def\_byte is qualified by the toggle ccc\_vld\_t.
- If there is no defining byte associated with the CCC, then the defining byte field are all zeros.
- The signal ccc\_vld\_t is asserted on the 2nd SCL clock during the address phase when the Slave address for the directed CCC is being transmitted by the I3C Master, provided there is no error associated with the CCC.
- The application understands if the directed CCC is meant for this Slave with the assertion of the ccc\_admtch\_dir\_t signal. This signal indicates that the ongoing directed CCC is targeted to this slave and gets asserted on the 9th SCL clock of the address phase if there is an address match and there is no error associated with the CCC.
- The application gets 8 SCL clock from the assertion of the ccc\_vld\_t to the point when ccc\_admtch\_dir\_t gets asserted to prepare for the directed CCC.
- ccc\_type and ccc\_def\_byte values are constant and preserve the previously received CCC until the receipt of the next CCC for the slave.
- If the received directed CCC has any payload data associated it is passed to the application through the ccc\_wr\_data channel along with the qualifier ccc\_wr\_data\_vld\_t.
- The ccc\_wr\_data and its qualifier ccc\_wr\_data\_vld\_t change at every 9th SCL clock (during T-Bit) after the address phase.
- The signals ccc\_wr\_data and its qualifier ccc\_wr\_data\_vld\_t toggle every time a ccc data byte is received. Hence, these signals are stable for a minimum of 9 SCL clocks.
- The signal ccc\_ongoing is asserted after 7e header the first bit of CCC is received and it is de-asserted either when a STOP or a 7e with a re-start comes on the I3C line.

## 2.13.2 Timing Diagram of Directed Write CCC with Defining Byte

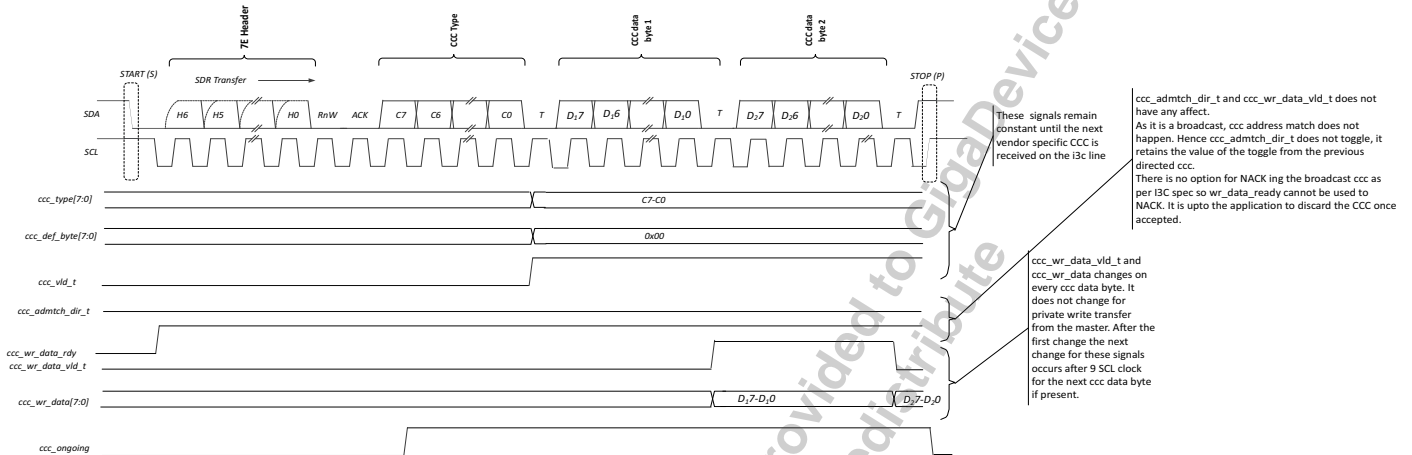
Figure 2-24 Timing Diagram of Directed Write CCC with Defining Byte



- Whenever a ccc which is not consumed internal to the slave is received, the ccc write channel is activated.
- The signal `ccc_vld_t` toggles on the 2nd SCL clock during the address phase when slave address for the directed CCC is transmitted by the I3C Master, provided there is no error associated with the CCC.
- The application understands if the directed CCC is meant for this slave with the assertion of the `ccc_admtch_dir_t` signal. This signal indicates that the ongoing directed CCC is targeted to this slave and gets asserted on the 9th SCL clock of the address phase if there is an address match and there is no error associated with the CCC.
- The application gets 8 SCL clock from the assertion of the `ccc_vld_t` to the point when `ccc_admtch_dir_t` gets asserted to prepare for the directed CCC.
- The value of the defining byte is presented on `ccc_def_byte` bus which is qualified by the `ccc_vld_t` toggle.
- The values for `ccc_type` and `ccc_def_byte` are constant and preserve the previously received CCC until the receipt of the next CCC for the slave.
- The `ccc_wr_data` and its qualifier `ccc_wr_data_vld_t` change at every 9th SCL clock (during T-Bit) after the address phase.
- The signals `ccc_wr_data` and its qualifier `ccc_wr_data_vld_t` toggle every time a ccc data byte is received. Hence, these signals are stable for a minimum of 9 SCL clocks. The signal `ccc_ongoing` is asserted when after 7e header the first bit of CCC is received and it is de-asserted either when a STOP or a 7e with a re-start comes on the I3C line.

### 2.13.3 Timing Diagram of Broadcast Write CCC without Defining Byte

Figure 2-25 Timing Diagram of Broadcast Write CCC without Defining Byte



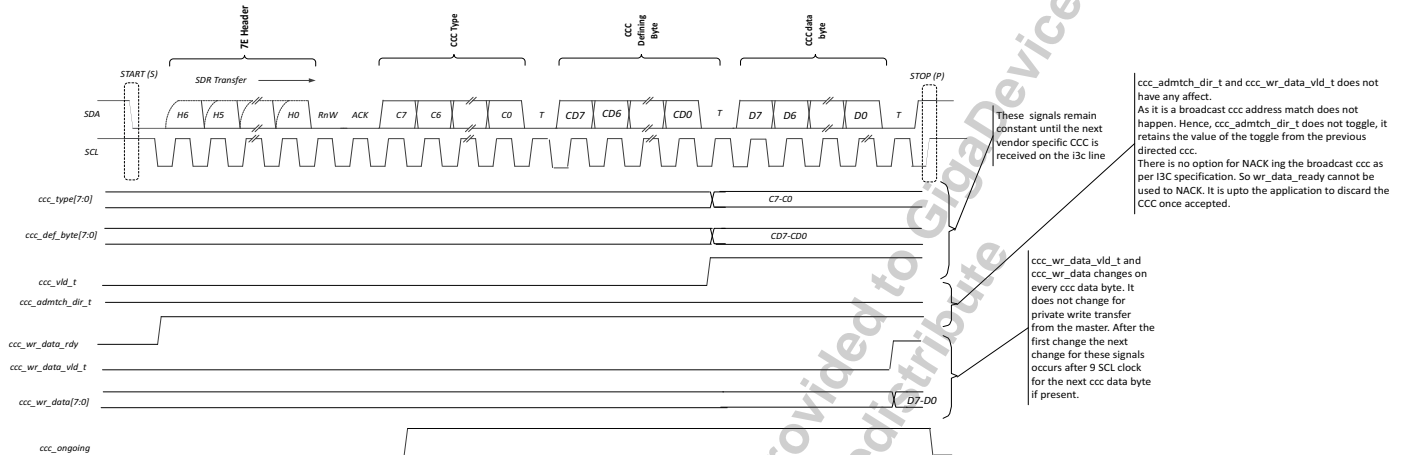
- Whenever a ccc which is not consumed internal to the slave is received, the ccc write channel is activated.
- The signals ccc\_type and ccc\_def\_byte are qualified by the toggle ccc\_vld\_t.
- If there is no defining byte associated with the CCC, then the defining byte field is all zeros.
- The signal ccc\_vld\_t is asserted on the 9th SCL clock during the CCC phase provided there is no error associated with the CCC.
- The values for ccc\_type and ccc\_def\_byte are constant and preserve the previously received CCC until the receipt of the next CCC for the slave.
- The broadcast CCC does not have any slave address associated. Hence, ccc\_admtch\_dir\_t signal does not toggle. It retains the value of the previous toggle which happened for a directed CCC.
- There is no provision for NACK'ing the broadcast CCC as per I3C specification. Hence, ccc\_wr\_data\_rdy does not affect the transfer. The application has to discard the CCC if it cannot accept the incoming broadcast CCC.
- If the received directed CCC has any payload data associated, it is passed to the application through the ccc\_wr\_data channel along with the qualifier ccc\_wr\_data\_vld\_t.
- The ccc\_wr\_data and its qualifier ccc\_wr\_data\_vld\_t change at every 9th SCL clock (during T-Bit) after the CCC type.
- The signals ccc\_wr\_data and its qualifier ccc\_wr\_data\_vld\_t toggle every time a ccc data byte is received. Hence, these signals are stable for a minimum of 9SCL clocks.
- The signal ccc\_ongoing is asserted when after 7e header the first bit of CCC is received and it is de-asserted either when a STOP or a re-start comes on the I3C line.



**Note** The CCC ongoing for special broadcast CCC like ENTDA de-asserts after the CCC is over on a STOP and not on restart and 7E.

## 2.13.4 Timing Diagram of Broadcast Write CCC with Defining Byte

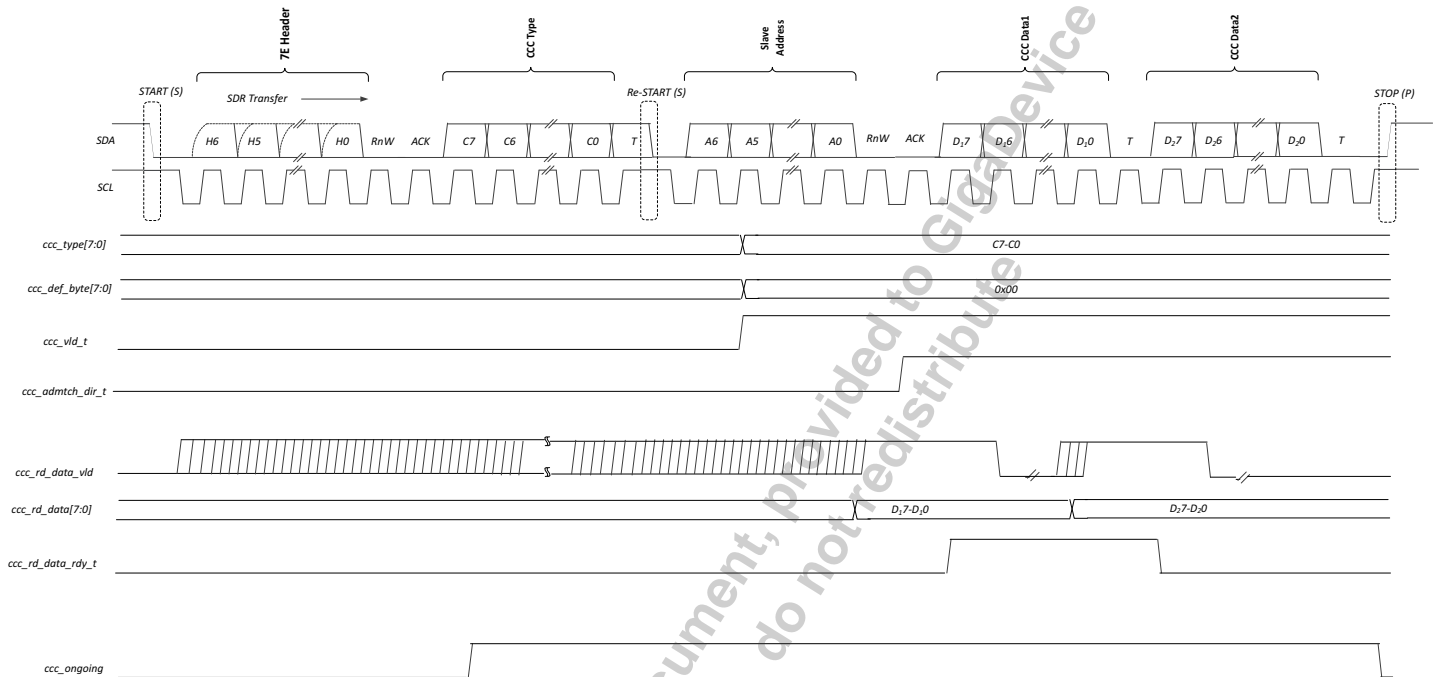
**Figure 2-26 Timing Diagram of Broadcast Write CCC with Defining Byte**



- Whenever a ccc which is not consumed internal to the slave is received, the ccc write channel is activated.
- The signal `ccc_vld_t` toggles on the 9th SCL clock of the defining byte of CCC.
- You must specify the broadcast CCC having defining byte during configuration.
- The broadcast CCCs which are not configured to have defining byte treats the defining byte as payload data and the defining byte is all zeros.
- The values for `ccc_type` and `ccc_def_byte` is constant and preserves the previously received CCC until the receipt of the next CCC for the slave.
- The broadcast CCC does not have any slave address associated. Hence, the `ccc_admtch_dir_t` signal does not toggle. It retains the value of the previous toggle which happened for a directed CCC.
- There is no provision for NACK'ing the broadcast CCC as per I3C specification hence `ccc_wr_data_rdy` does not have any affect on the transfer. The application has to discard the CCC if it cannot accept the incoming broadcast CCC.
- The `ccc_wr_data` and its qualifier `ccc_wr_data_vld_t` changes at every 9th SCL clock (during T-Bit) after the CCC defining byte.
- The signals `ccc_wr_data` and its qualifier `ccc_wr_data_vld_t` toggles every time a ccc data byte is received. Hence these signals are stable for a minimum of 9 SCL clocks.
- The signal `ccc_ongoing` is asserted when after 7e header the first bit of CCC is received and it is de-asserted either when a STOP or a re-start comes on the I3C line.

## 2.13.5 Timing Diagram of Directed Read CCC

Figure 2-27 Timing Diagram of Directed Read CCC



- Whenever a read ccc which is not consumed internal to the slave is received, the ccc read channel is activated.
- The signals ccc\_type and ccc\_def\_byte is qualified by the toggle ccc\_vld\_t.
- The signal ccc\_vld\_t is asserted on the 2nd SCL clock during the address phase when slave address for the directed CCC is transmitted by the I3C Master, provided there is no error associated with the CCC.
- The signal ccc\_rd\_data\_vld indicates the data present on ccc\_rd\_data is valid. This signal should be asserted before the 9th SCL Clock of the slave address phase. If this signal is not asserted before the 9th SCL clock of the slave address phase, the read CCC is NACKed by the controller. This signal along with the ccc\_rd\_data\_rdy\_t forms a handshake. This signal can be de-asserted after the toggle of ccc\_rd\_data\_rdy\_t.
- The application understands if the directed CCC is meant for this slave with the assertion of the ccc\_admtch\_dir\_t signal. This signal indicates that the ongoing directed CCC is targeted to this slave and gets asserted on the 9th SCL clock of the address phase if there is an address match and there is no error associated with the CCC.
- The application gets 8 SCL clock from the assertion of the ccc\_vld\_t to the point when ccc\_admtch\_dir\_t gets asserted to prepare for the directed CCC.
- The application data for the read ccc is provided in the ccc\_rd\_data input. This value should not change after ccc\_rd\_data\_vld has been asserted.
- If the directed read CCC is not for the slave which is indicated by ccc\_admtch\_dir\_t signal not toggling, the slave application must de-assert the ccc\_rd\_data\_vld after ccc\_ongoing gets de-asserted.
- ccc\_rd\_data\_vld should be de-asserted also when there is no more data present to be provided for the ongoing read CCC.

- The signal `ccc_rd_data_rdy_t` indicates the `ccc_rd_data` from the application has been accepted by the controller and the application can put forward the next data if required by the on-going CCC. The application must de-assert `ccc_rd_data_vld` after `ccc_rd_data_rdy_t` and again assert it before the 9th SCL clock of the ongoing data phase along with the new `ccc_rd_data`.
- The signal `ccc_ongoing` is asserted when after 7e header the first bit of CCC is received and it is de-asserted either when a STOP or a 7e with a re-start comes on the I3C line.

**Note**

For S5 error, the `ccc_ongoing` is de-asserted after the address match followed by restart for directed CCC which marks the end of CCC. For broadcast CCC with S5 error, the `ccc_ongoing` is de-asserted on restart. The `ccc_ongoing` is asserted for `enthdr ccc` when the first bit of `ccc` is received and it remains asserted until STOP is received on the line. The CCC `ongoing` is de-asserted for S0-S4 errors when it is decoded internally by the controller.

## 2.14 Support for CCCs for JESD403-1 Compliance

Table 2-10 Support for JESD403-1 Specific CCCs

CCC Type	CCC Code	Type	Mode
ENEC	0x00/0x80	B/D	In I3C mode only. Illegal in I2C mode
DISEC	0x01/0x81	B/D	In I3C mode only. Illegal in I2C mode
RSTDAA	0x06/0x86	B/D	In I3C mode only. It should be ignored in I2C mode
DEVCAP	0xE0	D	I3C mode only. Illegal in I2C mode
DEVCTRL	0x62	B	I2C mode and I3C mode
SETHID	0x61	B	I2C. Illegal in I3C mode
SETAASA	0x29	B	I2C. Ignored In I3C mode
GETSTATUS	0x90	D	I3C mode only



### Note

ENEC, DISEC, RSTDAA SETAASA and GETSTATUS CCCs are also applicable in I3C mode.

### 2.14.1 Handling of SETAASA CCC

The SETAASA CCC is applicable only in I2C mode. The SETAASA CCC does not affect the Dynamic Address of the Slave if it is received after the Slave Dynamic Address has already been set by any of the defined Address Assignment commands like SETDASA, ENTDA or SETAASA (issued earlier from the Master). This is a broadcast CCC. This CCC enables the Slave controller to make its static address as its dynamic address. The SETAASA CCC, once received by the Slave controller, makes its static address to its dynamic address. It changes its mode from I2C to I3C by asserting the dynamic address valid.

The SETAASA CCC is decoded internally for converting the static address to dynamic and the CCC type is passed to the application through the ccc\_type output port which is validated by the toggle signal ccc\_vld\_t. Refer [“Timing Diagram of Broadcast Write CCC without Defining Byte”](#) on page 75.

The PEC calculation is not applicable for this CCC as it is issued in I2C mode.

### 2.14.2 Handling of DEVCTRL CCC

The DEVCTRL CCC is applicable in both I2C and I3C mode. This is a broadcast CCC. This CCC is treated like vendor specific broadcast CCC without defining bytes. The CCC type and its associated payload including the device ID is passed to the application. The device ID is not matched internal to the controller. Refer to the timing diagram for Broadcast Write CCC without defining byte [“Timing Diagram for Broadcast Write CCC without Defining Byte”](#).

In I2C mode of operation, PEC calculation is disabled. If PEC is enabled in I3C mode of operation, the PEC byte is passed out of the controller through the ccc write data channel qualified with the toggle ccc\_wr\_data\_vld\_t.



Refer [“Support for PEC in Private Transfers for JESD403-1 Compliance”](#) on page 58 for PEC calculation and [“Timing Diagram of Broadcast Write CCC with PEC Error”](#) on page 84 for the timing diagram.

### 2.14.3 Handling of SETHID CCC

The SETHID CCC is applicable only in I2C mode. If received in I3C mode, the Slave passes this CCC through the CCC channel. The Master should not issue this CCC in I3C mode of operation. This is a broadcast CCC. This CCC is treated like vendor specific broadcast CCC without defining bytes. The CCC type and its associated payload is passed to the application. Refer [“Timing Diagram of Broadcast Write CCC without Defining Byte”](#) on page 75”.

On receiving the SETHID CCC through the CCC channel, the Slave-Lite application should do the following:

- De-assert the static address enable to ascertain the static address is not sampled by the controller.
- Change the static address as per the HID value received from the SETHID CCC
- Assert the static address enable to enable the controller to sample the new static address.

The Slave application should change the static address and enable the static address enable input. If enabling the static address from the Slave application is late and the next immediate transfer from the master is directed to the Slave with the new static address, there can be a possibility that the immediate transfer is NACK'ed.

### 2.14.4 Handling of ENEC, DISEC, and RSTDAA CCC

These CCCs are applicable only in I3C mode. These are handled internal to the controller and not passed to the application through the ccc channel. CCCs support PEC only in I3C mode. Refer to the timing diagram given in sections [“Timing Diagram of Broadcast Write CCC with PEC Error”](#) on page 84 and [“Timing Diagram of Directed Write CCC with PEC Error”](#) on page 84 for understanding the behavior of the PEC error signal and its qualifier.

### 2.14.5 Handling of GETSTATUS CCC

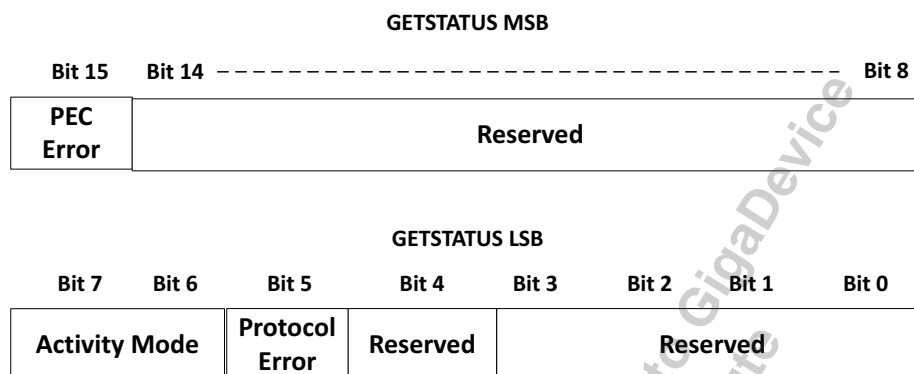
The GETSTATUS CCC is a directed CCC applicable only in I3C mode.

- GETSTATUS CCC

The slave controller consumes this CCC internally and sends out the response as associated data in the format given in [Figure 2-28](#). This CCC is not given out to the ccc channel.

The data associated with the GETSTATUS is shown in [Figure 2-28](#).



**Figure 2-28 Data Associated with GETSTATUS CCC for Format 1**

If PEC is enabled for this CCC, it has the structure as given in the section [“Directed CCC write with PEC Enabled”](#) on page 83.

### 2.14.6 Handling of DEVCAP CCC

It is a directed CCC allowed in only I3C mode. If issued in I2C mode, the application can NACK it by de-asserting `ccc_rd_data_vld`. This CCC is not consumed internal to the IP, but passed on the CCC channel. The application should drive the `ccc_rd_data`, `ccc_rd_data_vld` appropriately with CCC data. Refer [“Timing Diagram of Directed Write CCC without Defining Byte”](#) on page 73. If PEC is enabled, the PEC byte associated with this CCC is passed to the application through the `ccc_def_byte` port. The PEC is decoded internally and `slv_pec_err` toggles if PEC error is present. The controller NACK's the slave address for this directed CCC if PEC error is detected. Refer [“Timing Diagram of Directed Write CCC with PEC Error”](#) on page 84.

## 2.15 Handling of Unsupported CCC

The Slave-Lite controller accepts all CCCs from the Master. The CCCs meant for function which the controller does not support is accepted and passed through the ccc write channel to the application (For example, MLANE, RSTGRPA).

If the CCC is a directed write CCC, the application can NACK the CCC by de-asserting the `ccc_wr_data_rdy`. The application has to decode the CCC type and if the CCC type is of un-supported CCC, `ccc_wr_data_rdy` should be de-asserted.

If `ccc_wr_data_rdy` is not de-asserted, the CCC ACK'ed by the controller. For directed read CCC, the application can NACK the CCC by de-asserting the `ccc_rd_data_vld`. If `ccc_rd_data_vld` is not de-asserted, the CCC is ACKED.

[Table C-1](#) lists the CCC types which uses the CCC channel. The CCCs consumed internally are not passed through this channel. The CCCs passed through the channel might be targeted for functions not supported by the controller.

## 2.16 PEC Support for CCC for JESD403-1 Compliance

### 2.16.1 Broadcast CCC write with PEC Enabled

The Master appends the PEC byte at the end of broadcast write CCC transfers if PEC is enabled. The PEC is calculated only on the CCC code and optional data (and not for 7E/W). The Slave computes the CRC8 while it receives CCC code and data, and compare the CRC8 value with the incoming PEC byte. The slave flags a PEC error if internally computed CRC8 value and received PEC byte is mismatching.

**Figure 2-29 Broadcast CCC write with PEC Enabled**

START (S/Sr)	Address (0x7E,w)	Broadcast CCC Code (CCC,T)	Data (Optional)	PEC Byte (CRC Byte,T)	RESTART (Sr)/ STOP (P)
-----------------	---------------------	-------------------------------	--------------------	--------------------------	---------------------------

If PEC error is detected the slave NACK's all transfers until Stop (P) is detected.



#### Note

PEC is supported only for CCCs that are mandatory for SidebandBus devices.

### 2.16.2 Directed CCC write with PEC Enabled

The Master appends the PEC byte at the end of CCC code and at the end of directed CCC data payload. The PEC byte at the end of CCC code is computed only on CCC code. The Slave also computes the CRC8 on the received CCC code and compares the CRC8 value with the received PEC byte. The slave flags PEC error if there is a mismatch between internally computed CRC8 value and received PEC byte. If PEC error is detected, the Slave also NACK's the Slave Address for the same CCC and NACK's all transfers before Stop.

The PEC byte at the end of CCC payload data is computed on both the slave address and data payload. The Slave also computes CRC8 on the received address and data payload and discards the data if PEC error is detected.

**Figure 2-30 Directed CCC write with PEC Enabled**

START (S/Sr)	Address (0x7E,w)	Directed CCC Code (CCC,T)	PEC Byte (CRC Byte,T)	RESTART (Sr)	...
-----------------	---------------------	------------------------------	--------------------------	-----------------	-----

...	Slave Address (SA,w/r)	Data Payload (DB,T)	PEC Byte (CRC Byte,T)	STOP (P/Sr)
-----	---------------------------	------------------------	--------------------------	----------------

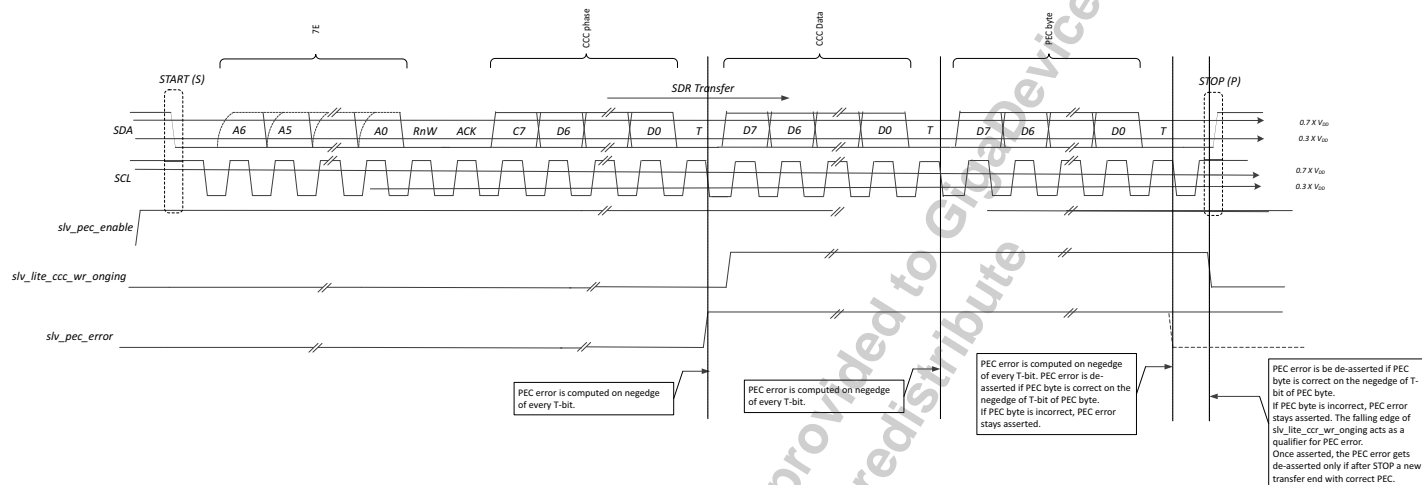


#### Note

PEC is supported only for CCCs that are mandatory for SidebandBus devices.

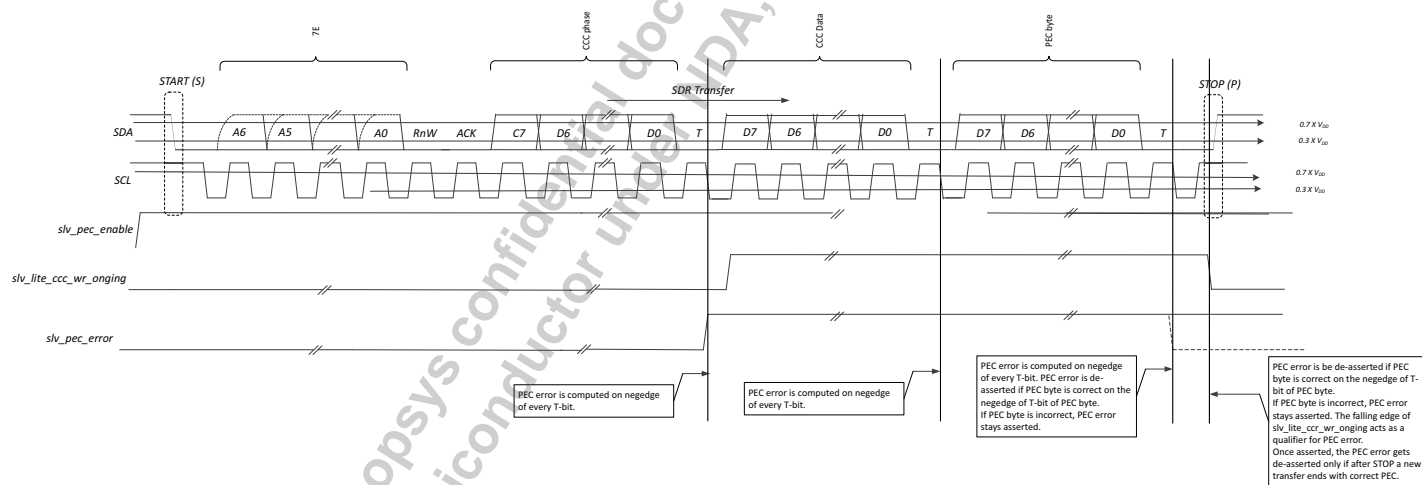
## 2.16.3 Timing Diagram of Broadcast Write CCC with PEC Error

Figure 2-31 Broadcast CCC write with PEC Error

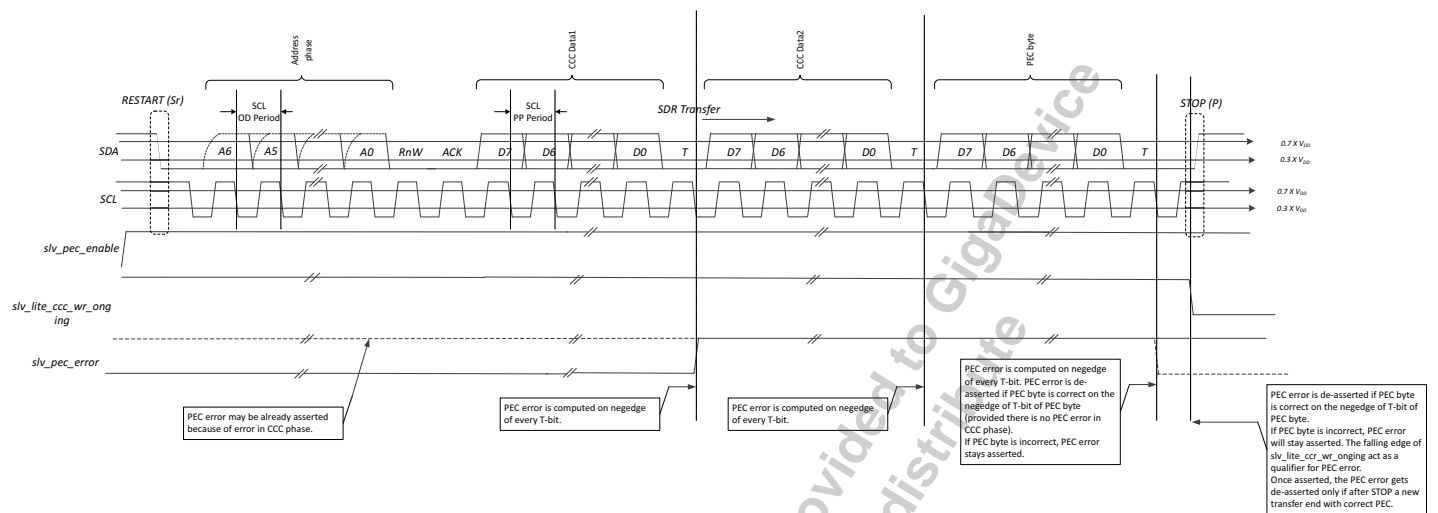


## 2.16.4 Timing Diagram of Directed Write CCC with PEC Error

Figure 2-32 Directed CCC Write with PEC error



# CCC Continued



## 2.17 Bus Reset

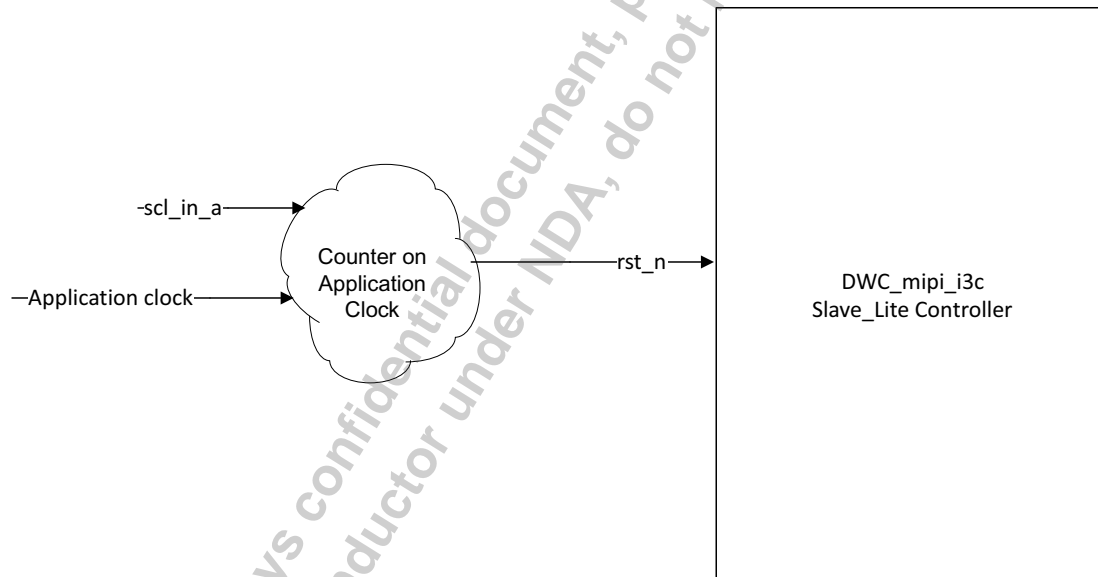
To prevent a malfunctioning device from locking up the I2C or I3C bus, a reset mechanism is defined in the JESD403-1 specification. This reset is a timeout-based reset. The reset should be generated by the Slave-Lite application as the controller which is running on SCL clock cannot keep count of the time for which SCL is low.

The application can have a counter running on application clock to count the low period of the SCL. This counter gets reset whenever there is any event on the SCL line. The expiry of the counter triggers the reset assertion.

The DWC\_mipi\_i3c Slave-Lite controller expects the application generated reset to be connected to the hardware reset pin “rst\_n”. Following are the requirements from the Slave-Lite controller for the reset assertion and de-assertion from the Slave-Lite application.

- The DWC\_mipi\_i3c controller in Slave-Lite mode expects the generated reset to have a negative polarity (to reset the controller the input “rst\_n” should be made low).

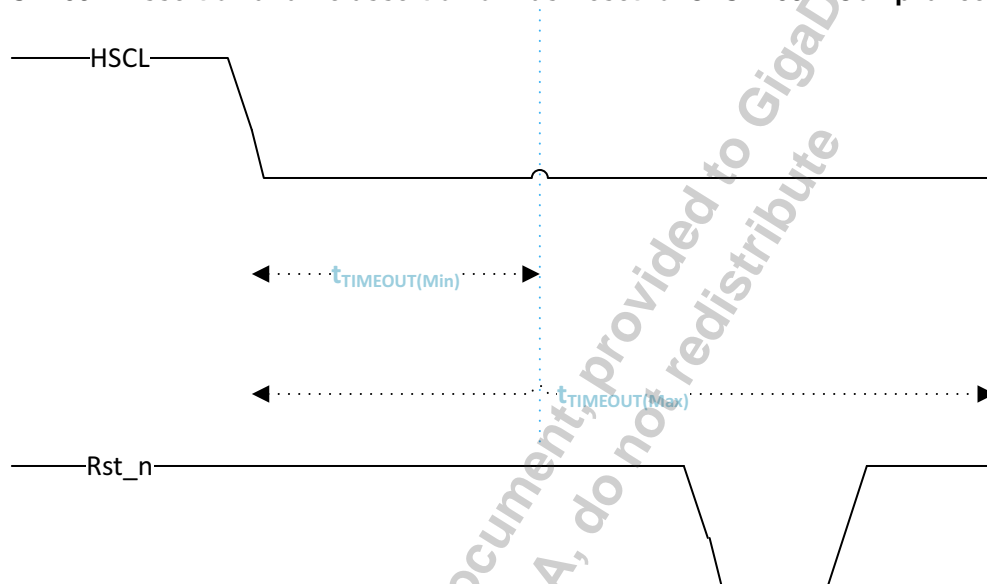
**Figure 2-33 Application Generated Reset Connected to rst\_n for Bus Reset**



- The reset assertion and de-assertion should be done when the SCL clock is low, that is, when there is no event on the SCL line. In other words, the controller expects both assertion and de-assertion to be asynchronous when no clock (SCL) is present. To achieve this condition a possible flow has been suggested as follows:
  - As per the JESD403-1 specification, the SCL clock Low time must be greater than or equal to  $t_{TIMEOUT(Max)}$  where  $t_{TIMEOUT(Max)}$  is 50 ms. The JESD403-1 specification also states that the I2C or I3C bus does not get reset if the SCL clock input Low time is less than  $t_{TIMEOUT(Min)}$  where  $t_{TIMEOUT(Min)}$  is 10 ms.
  - To abide by the JESD403-1 given specification and the requirement of assertion and de-assertion of the reset (input “rst\_n”) when there is no SCL, the Slave-Lite application can characterize the reset behavior as follows:
    - Assert the reset after the  $t_{TIMEOUT(Min)}$  period at 30ms.

- The input pin “rst\_n” should be kept low for at least two SCL clock period duration. (Assuming SCL clock frequency of 12.5 MHz the low period for the reset is 160 ns).
- The de-assertion of the reset should happen between tTIMEOUT(Min) and tTIMEOUT(Max) (after duration of 40ms) to ensure the reset de-assertion happens when the SCL clock is not present.

**Figure 2-34 JESD403-1 Assertion and De-assertion of Bus Reset for JESD403-1 Compliance**



### 2.17.1 Slave-Lite State after Bus Reset

After the bus reset is applied the state of the Slave-Lite controller is as given in [Table 2-11](#).

**Table 2-11 Slave-Lite Status after Bus Reset**

Internal Registers		
Registers Holding CCC status		
Activity State register holding the value of ENTAS	The register gets cleared and has the initial value of all zero.	The corresponding output ping for activity_state drives zero.
The IBI status holding the value of the ENEC and STATUS	This register is cleared indicating IBI is disabled from the master. A separate ENEC CCC to enable the IBI is required after reset.	
MRL and MWL value from SETMRL and SETMWL	This register is cleared and has the value of zero.	The corresponding output port of MRL and MWL reflects all zero.
activity mode, pending interrupt and internal error flag for protocol error for GETSTATUS CCCC	These registers are cleared.	The input port of activity mode and pending interrupt is expected to be zero.
Operating Mode		

Mode change to I2C	The internal register holding the current mode of the controller resets to I2C mode	
Sub-address and Dynamic Address.		
Dynamic address	The internal register holding the dynamic address of the controller is reset.	The output pin indicating dynamic address is valid is zero.
sub address	The internal register holding the last sub address value is reset	The sub-address output port is all zero.
<b>Toggle Hand Shake Signal for CCC and Private Read/Write channels</b>		
Toggle Signals for CCC, Private Read/Write	All the internal registers driving the handshake toggle signals are cleared	The corresponding output pins of all toggle signals are reset to 0
<b>Expected Driving of Input Signal of the Controller</b>		
slv_par_err_dis	This input should be driven to 1	
slv_par_err_ext	This input should be driven to 0	
static_address_en	This input is required to be driven to 0	
pec_err_en	This input should be driven to 0	
slv_pec_err_ext	This input should be driven to 0	
rd_data_vld	This input should be driven to 0	
wr_data_rdy	This input should be driven to 0	



# 3

## Parameter Descriptions

This chapter details all the configuration parameters. **You can use the coreConsultant GUI configuration reports to determine the complete configuration state of the controller.** Some expressions might refer to TCL functions or procedures (sometimes identified as `<functionof>`) that coreConsultant uses to make calculations. The exact formula used by these TCL functions is not provided in this chapter. However, when you configure the controller in coreConsultant, all TCL functions and parameters are evaluated completely; and the resulting values are displayed where appropriate in the coreConsultant GUI reports.

The parameter descriptions in this chapter include the **Enabled:** attribute which indicates the values required to be set on other parameters before you can change the value of this parameter.

These tables define all of the configuration options for this component.

- [“Basic Configuration Parameters” on page 90](#)
- [“Slave-lite Configuration Parameters” on page 91](#)
- [“Slave Configuration Parameters” on page 92](#)

### 3.1 Basic Configuration Parameters

**Table 3-1 Basic Configuration Parameters**

Label	Description
HDR Support	
HDR-DDR	<p>Configures DWC_mipi_i3c to have high data rate - Double Data Rate Mode support. If this parameter is selected, DWC_mipi_i3c transfers the data on both edge transitions of the SCL for generation of high data rate.</p> <p><b>Values:</b>0, 1</p> <p><b>Default Value:</b>0</p> <p><b>Enabled:</b>Always</p> <p><b>Parameter Name:</b>IC_SPEED_HDR_DDR</p>
PEC Support	
PEC Support Enable	<p>Configures DWC_mipi_i3c to support Packet Error Check (PEC) in SDR transfers. If this parameter is selected, DWC_mipi_i3c (In Master-only and Slave-lite configurations) will have capability to generate and validate PEC Byte in</p> <ul style="list-style-type: none"> <li>■ SDR Broadcast and Directed CCC Transfers,</li> <li>■ SDR Private Write and Read Transfers and</li> <li>■ SDR IBI Transfers</li> </ul> <p>as per the JEDEC Sideband specification.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>■ false (0)</li> <li>■ true (1)</li> </ul> <p><b>Default Value:</b>false</p> <p><b>Enabled:</b>((IC_DEVICE_ROLE==1)    (IC_DEVICE_ROLE==5)) &amp;&amp; (IC_HAS_HCI==0)</p> <p><b>Parameter Name:</b>IC_HAS_PEC</p>

## 3.2 Slave-lite Configuration Parameters

**Table 3-2 Slave-lite Configuration Parameters**

Label	Description
Slave Lite Application Interface Settings	
Transmit side Bus Width of Application Interface	<p>Default Transmit side Application Interface Width of DWC_mipi_i3c_slv_lite.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>■ 8bit (8)</li> <li>■ 32bit (32)</li> </ul> <p><b>Default Value:</b>((IC_DEVICE_ROLE&gt;1) &amp;&amp; (IC_DEVICE_ROLE&lt;5)) ? 32:8</p> <p><b>Enabled:</b>IC_DEVICE_ROLE==5</p> <p><b>Parameter Name:</b>IC_SLV_INTERFACE_TX_WIDTH</p>
Receive side Bus Width of Application Interface	<p>Default Receive side Application Interface Width of DWC_mipi_i3c_slv_lite.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>■ 8bit (8)</li> <li>■ 16bit (16)</li> </ul> <p><b>Default Value:</b>((IC_DEVICE_ROLE&gt;1) &amp;&amp; (IC_DEVICE_ROLE&lt;5)) ? 16 : (IC_SLV_INTERFACE_TX_WIDTH==32 ? 16:8)</p> <p><b>Enabled:</b>0</p> <p><b>Parameter Name:</b>IC_SLV_INTERFACE_RX_WIDTH</p>

### 3.3 Slave Configuration Parameters

**Table 3-3 Slave Configuration Parameters**

Label	Description
Bus Characteristic Register Configuration	
Bridge Device	<p>Specifies whether DWC_mipi_i3c is a bridge device or not.</p> <ul style="list-style-type: none"> <li>■ 0: Not a bridge device</li> <li>■ 1: Bridge device</li> </ul> <p><b>Note:</b> Bridge device is not supported.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>■ false (0)</li> <li>■ true (1)</li> </ul> <p><b>Default Value:</b>false</p> <p><b>Enabled:</b>0</p> <p><b>Parameter Name:</b>IC_SLV_BRIDGE</p>
Offline Capable	<p>Specifies whether DWC_mipi_i3c is offline capable or not.</p> <ul style="list-style-type: none"> <li>■ 0: Not offline capable</li> <li>■ 1: Offline capable</li> </ul> <p><b>Note:</b> Offline capability is not supported</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>■ false (0)</li> <li>■ true (1)</li> </ul> <p><b>Default Value:</b>false</p> <p><b>Enabled:</b>0</p> <p><b>Parameter Name:</b>IC_SLV_OFFLINE_CAP</p>
IBI Support	<p>Specifies whether DWC_mipi_i3c supports Slave In-Band Interrupt or not.</p> <ul style="list-style-type: none"> <li>■ 0: Slave In-Band Interrupt is not supported</li> <li>■ 1: Slave In-Band Interrupt is supported</li> </ul> <p>If this parameter is selected the DWC_mipi_i3c Slave will support both SIR and Hot-Join. The configuration parameter for Hot-join, IC_SLV_HJ, will be automatically set to 1, if IC_SLV_IBI is set.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>■ false (0)</li> <li>■ true (1)</li> </ul> <p><b>Default Value:</b>false</p> <p><b>Enabled:</b>IC_DEVICE_ROLE&gt;1</p> <p><b>Parameter Name:</b>IC_SLV_IBI</p>

Label	Description
IBI with Data Support	<p>Specifies whether DWC_mipi_i3c supports Slave In-Band interrupt with data or not.</p> <ul style="list-style-type: none"> <li>0: Slave In-band interrupt with data is not supported</li> <li>1: Slave In-band interrupt with data is supported</li> </ul> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>false (0)</li> <li>true (1)</li> </ul> <p><b>Default Value:</b>false</p> <p><b>Enabled:</b>IC_DEVICE_ROLE==5</p> <p><b>Parameter Name:</b>IC_SLV_IBI_DATA</p>
Max Data Speed Limitation	<p>Specifies whether or not DWC_mipi_i3c has maximum data speed limitation. Note that this bit can be programmed/overwritten by software. If it is unclear whether or not your design has Maximum Data Speed Limitation, select "Include programming for maxRd, maxWr and tsc0" to have the flexibility of providing these values through programmable registers.</p> <ul style="list-style-type: none"> <li>0: No limitation</li> <li>1: Limitation</li> </ul> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>false (0)</li> <li>true (1)</li> </ul> <p><b>Default Value:</b>false</p> <p><b>Enabled:</b>IC_DEVICE_ROLE&gt;1</p> <p><b>Parameter Name:</b>IC_SLV_DATA_SPEED_LIMIT</p>
Defining Byte identifier for Vendor Specific Broadcast CCC	
Vendor Specific Broadcast CCC	<p>Vendor Specific Broadcast CCC having Defining byte. The controller supports only one vendor specific CCC with defining byte. This is required to distinguish the defining byte from the CCC payload data. The defining byte will be passed to the application through the defining byte output port with the qualifier ccc_vld_t.</p> <p><b>Values:</b>0x65, ..., 0x7f</p> <p><b>Default Value:</b>0x65</p> <p><b>Enabled:</b>IC_DEVICE_ROLE==5</p> <p><b>Parameter Name:</b>IC_SLV_BCST_VEND_CCC</p>

Label	Description
Max Data Speed Configuration	
Maximum Sustained Write Data Rate	<p>Specifies the Maximum Sustained Data Rate for non-CCC messages sent by Master Device to DWC_mipi_i3c Slave device.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>■ fSCL Max(12.5 MHz) (0)</li> <li>■ 8MHz (1)</li> <li>■ 6MHz (2)</li> <li>■ 4MHz (3)</li> <li>■ 2MHz (4)</li> <li>■ Reserved_5 (5)</li> <li>■ Reserved_6 (6)</li> <li>■ Reserved_7 (7)</li> </ul> <p><b>Default Value:</b>fSCL Max(12.5 MHz)</p> <p><b>Enabled:</b>IC_SLV_MXDS_PROG==0</p> <p><b>Parameter Name:</b>IC_SLV_MXDS_MAX_WR_SPEED</p>
Maximum Sustained Read Data Rate	<p>Specifies the Maximum Sustained Data Rate for non-CCC messages sent by DWC_mipi_i3c Slave Device to Master device.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>■ fSCL Max(12.5 MHz) (0)</li> <li>■ 8MHz (1)</li> <li>■ 6MHz (2)</li> <li>■ 4MHz (3)</li> <li>■ 2MHz (4)</li> <li>■ Reserved_5 (5)</li> <li>■ Reserved_6 (6)</li> <li>■ Reserved_7 (7)</li> </ul> <p><b>Default Value:</b>fSCL Max(12.5 MHz)</p> <p><b>Enabled:</b>IC_SLV_MXDS_PROG==0</p> <p><b>Parameter Name:</b>IC_SLV_MXDS_MAX_RD_SPEED</p>

Label	Description
Clock to Data Turnaround Time	<p>Specifies the clock to data turnaround time (Tsc0 parameter) of DWC_mipi_i3c Slave device.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>■ 8ns (0)</li> <li>■ 9ns (1)</li> <li>■ 10ns (2)</li> <li>■ 11ns (3)</li> <li>■ 12ns (4)</li> <li>■ Reserved_5 (5)</li> <li>■ Reserved_6 (6)</li> <li>■ Reserved_7 (7)</li> </ul> <p><b>Default Value:</b>8ns  <b>Enabled:</b>IC_SLV_MXDS_PROG==0  <b>Parameter Name:</b>IC_SLV_MXDS_CLK_DATA_TURN</p>
Maximum Read Turnaround Time	<p>Specifies the maximum read turnaround time (in microseconds (us)) of DWC_mipi_i3c Slave Lite.</p> <p><b>Values:</b>0x0, ..., 0xfffff</p> <p><b>Default Value:</b>0x0  <b>Enabled:</b>Always  <b>Parameter Name:</b>IC_SLV_MXDS_MAX_RD_TURN</p>
Default Values	
MWL Value	<p>Default Maximum Write Length value of DWC_mipi_i3c. Sets the reset value of Maximum Write Length register of DWC_mipi_i3c. This value is returned by the DWC_mipi_i3c Slave if the Master sends a GETMWL CCC. The Maximum Write Length register is overwritten by a new MWL value if the Master sends SETMWL CCC.</p> <p><b>Values:</b>0x0, ..., 0xffff</p> <p><b>Default Value:</b>0xff  <b>Enabled:</b>IC_DEVICE_ROLE&gt;1  <b>Parameter Name:</b>IC_SLV_DFLT_MWL</p>
MRL Value	<p>Default Maximum Read Length value of DWC_mipi_i3c. Sets the reset value of Maximum Read Length register of DWC_mipi_i3c. This value is returned by the DWC_mipi_i3c Slave if the Master sends a GETMRL CCC. The Maximum Read Length register is overwritten by a new MRL value if the Master sends SETMRL CCC.</p> <p><b>Values:</b>0x0, ..., 0xffff</p> <p><b>Default Value:</b>0xff  <b>Enabled:</b>IC_DEVICE_ROLE&gt;1  <b>Parameter Name:</b>IC_SLV_DFLT_MRL</p>

Label	Description
IBI Payload Size	<p>Default IBI Payload Size of DWC_mipi_i3c. Sets the reset value of IBI payload size register of DWC_mipi_i3c. This value is returned by the DWC_mipi_i3c Slave as IBI Payload Size if the Master sends a GETMRL CCC. The IBI payload size register is overwritten by a new payload size value if the Master sends SETMRL CCC. IBI payload size register is supported only if the Slave supports IBI with data.</p> <p><b>Values:</b>0x0, ..., 0xff</p> <p><b>Default Value:</b>0x1</p> <p><b>Enabled:</b>IC_SLV_IBI_DATA==1</p> <p><b>Parameter Name:</b>IC_SLV_DFLT_IBI_PAYLOAD_SIZE</p>



## 4

## Signal Descriptions

This chapter details all possible I/O signals in the IP. For configurable IP titles, your actual configuration might not contain all of these signals.

Inputs are on the left of the signal diagrams; outputs are on the right.

**Attention: For configurable IP titles, do not use this document to determine the exact I/O footprint of the controller. It is for reference purposes only.**

When you configure the controller in coreConsultant, you must access the I/O signals for your actual configuration at workspace/report/IO.html or workspace/report/IO.xml after you have completed the report creation activity. That report comes from the exact same source as this chapter but removes all the I/O signals that are not in your actual configuration. This does not apply to non-configurable IP titles. In addition, all parameter expressions are evaluated to actual values. Therefore, the widths might change depending on your actual configuration.

Some expressions might refer to TCL functions or procedures (sometimes identified as **<functionof>**) that coreConsultant uses to make calculations. The exact formula used by these TCL functions is not provided in this chapter. However, when you configure the controller in coreConsultant, all TCL functions and parameters are evaluated completely; and the resulting values are displayed where appropriate in the coreConsultant GUI reports.

In addition to describing the function of each signal, the signal descriptions in this chapter include the following information:

- **Active State:** Indicates whether the signal is active high or active low. When a signal is not intended to be used in a particular application, then this signal needs to be tied or driven to the inactive state (opposite of the active state).
- **Registered:** Indicates whether or not the signal is registered directly inside the IP boundary without intervening logic (excluding simple buffers). A value of *No* does not imply that the signal is not synchronous, only that there is some combinatorial logic between the signal's origin or destination register and the boundary of the controller. A value of *N/A* indicates that this information is not provided for this IP title.
- **Synchronous to:** Indicates which clocks in the IP sample this input (drive for an output). This clock might not be the same as the clock that your application logic should use to clock (sample/drive) this pin. For more details, consult the clock section in the databook.
- **Exists:** Name of configuration parameter that populates this signal in your configuration.

The I/O signals are grouped as follows:

- [“Debug Interface Signals”](#) on page 99
- [“I3C Interface Signals”](#) on page 100
- [“Slave Interface Signals”](#) on page 103
- [“Scan Interface Signals”](#) on page 117

Note: The Debug port interface is not validated in this release.

Synopsys confidential document, provided to GigaDevice  
Semiconductor under NDA, do not redistribute

## 4.1 Debug Interface Signals



**Table 4-1**      **Debug Interface Signals**

Port Name	I/O	Description
debug_port[(IC_DBG_DW-1):0]	O	<p>Debug port On chip debug signal as follows:</p> <p>debug[47:33] = Reserved</p> <p>■ Slave Lite mode debug Signals            debug[7:0] = Dynamic Address of the Slave.            debug[16:8] = Command Code Received.            debug[17] = Internal Error Latched untill GETSTATUS.</p> <p><b>Exists:</b>(IC_HAS_DEBUG_PORTS==1)  <b>Synchronous To:</b>(IC_DEVICE_ROLE==5) ? "scl_in_a" :            ((IC_SLAVE_APB_EN==1) &amp;&amp; (IC_MAIN_MASTER_EN==0) ? "pclk" :            "core_clk")  <b>Registered:</b>No  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>

## 4.2 I3C Interface Signals

scl\_in\_a\_n  
 sda\_in\_a\_n  
 scl\_in\_a  
 sda\_in\_a

-sda\_oe  
 -scl\_out  
 -sda\_out  
 -scl\_oe

**Table 4-2 I3C Interface Signals**

Port Name	I/O	Description
scl_in_a_n	I	<p>Incoming inverted SCL Applicable only in Slave mode of operation.</p> <p><b>Exists:</b> (IC_DEVICE_ROLE &gt; 1)    ((IC_DEVICE_ROLE == 1) &amp;&amp; (IC_SPEED_HDR_DDR == 1) &amp;&amp; (IC_SDA_SAMPLING_IN_SCL == 1))</p> <p><b>Synchronous To:</b> None</p> <p><b>Registered:</b> No</p> <p><b>Power Domain:</b> SINGLE_DOMAIN</p> <p><b>Active State:</b> High</p>
sda_in_a_n	I	<p>Incoming Inverted SDA Applicable only in Slave mode of operation.</p> <p><b>Exists:</b> (IC_DEVICE_ROLE &gt; 1)</p> <p><b>Synchronous To:</b> (IC_SLAVE_APB_EN == 1) &amp;&amp; (IC_SPEED_HDR_EN == 1) ? "scl_in_a, scl_in_a_n" : (IC_SLAVE_APB_EN == 1) &amp;&amp; (IC_SPEED_HDR_EN == 0) ? "scl_in_a" : "None"</p> <p><b>Registered:</b> No</p> <p><b>Power Domain:</b> SINGLE_DOMAIN</p> <p><b>Active State:</b> High</p>
sda_oe	O	<p>Outgoing SDA Pad enable This signal is used to enable the pad to switch between OD(0) and PP(1) mode.</p> <p><b>Exists:</b> Always</p> <p><b>Synchronous To:</b> ((IC_DEVICE_ROLE &lt; 4) &amp;&amp; (IC_DEVICE_ROLE &gt; 1)) ? "core_clk, scl_in_a, scl_in_a_n" : ((IC_DEVICE_ROLE &gt; 3) ? "scl_in_a, scl_in_a_n" : "core_clk")</p> <p><b>Registered:</b> (IC_DEVICE_ROLE &gt; 1) ? "No" : "Yes"</p> <p><b>Power Domain:</b> SINGLE_DOMAIN</p> <p><b>Active State:</b> High</p>

Port Name	I/O	Description
scl_out	O	<p>SCL output Signal This signal is an input to the SCL PAD Driver.</p> <p><b>Exists:</b>((IC_OPEN_DRAIN_CLASS_PULLUP==1) &amp;&amp; (IC_DEVICE_ROLE&lt;4))    ((IC_DEVICE_ROLE&gt;1) &amp;&amp; (IC_SPEED_HDR_TS==1))</p> <p><b>Synchronous To:</b>((IC_DEVICE_ROLE&gt;1) &amp;&amp; (IC_DEVICE_ROLE&lt;4)) &amp;&amp; (IC_SPEED_HDR_TS==1) ? "core_clk,hdr_tx_clk" : ((IC_DEVICE_ROLE==4) &amp;&amp; (IC_SPEED_HDR_TS==1)) ? "hdr_tx_clk" : "core_clk"</p> <p><b>Registered:</b>((IC_DEVICE_ROLE&gt;1) &amp;&amp; (IC_DEVICE_ROLE&lt;4)) ? "No" : "Yes"</p> <p><b>Power Domain:</b>SINGLE_DOMAIN</p> <p><b>Active State:</b>High</p>
sda_out	O	<p>SDA output Signal This signal is an input to the SDA PAD Driver.</p> <p><b>Exists:</b>(IC_OPEN_DRAIN_CLASS_PULLUP==1)    (IC_DEVICE_ROLE==5)</p> <p><b>Synchronous To:</b>((IC_DEVICE_ROLE&gt;1) &amp;&amp; (IC_DEVICE_ROLE&lt;4)) ? "core_clk,scl_in_a,scl_in_a_n" : ((IC_DEVICE_ROLE&gt;3) ? "scl_in_a,scl_in_a_n" : "core_clk")</p> <p><b>Registered:</b>(IC_DEVICE_ROLE&gt;1) ? "No" : "Yes"</p> <p><b>Power Domain:</b>SINGLE_DOMAIN</p> <p><b>Active State:</b>High</p>
scl_oe	O	<p>Outgoing SCL Pad enable This signal is used to enable the Pad to switch between OD(0) and PP(1) mode.</p> <p><b>Exists:</b>(IC_DEVICE_ROLE&lt;4)    (IC_SPEED_HDR_TS==1)</p> <p><b>Synchronous To:</b>((IC_DEVICE_ROLE&gt;1) &amp;&amp; (IC_DEVICE_ROLE&lt;4)) &amp;&amp; (IC_SPEED_HDR_TS==1) ? "core_clk,hdr_tx_clk" : ((IC_DEVICE_ROLE==4) &amp;&amp; (IC_SPEED_HDR_TS==1)) ? "hdr_tx_clk" : ((IC_DEVICE_ROLE==1) &amp;&amp; (IC_SPEED_HDR_TS==1) &amp;&amp; (IC_OPEN_DRAIN_CLASS_PULLUP_EN==1)) ? "core_clk" : ((IC_DEVICE_ROLE==1) &amp;&amp; (IC_SPEED_HDR_TS==0) &amp;&amp; (IC_OPEN_DRAIN_CLASS_PULLUP_EN==0)) ? "core_clk" : "None"</p> <p><b>Registered:</b>((IC_DEVICE_ROLE&gt;1) &amp;&amp; (IC_DEVICE_ROLE&lt;4)) ? "No" : "Yes"</p> <p><b>Power Domain:</b>SINGLE_DOMAIN</p> <p><b>Active State:</b>High</p>
scl_in_a	I	<p>Incoming SCL Input SCL Signal from the SCL Pad Driver.</p> <p><b>Exists:</b>Always</p> <p><b>Synchronous To:</b>(IC_DEVICE_ROLE==1) ? "core_clk" : "None"</p> <p><b>Registered:</b>No</p> <p><b>Power Domain:</b>SINGLE_DOMAIN</p> <p><b>Active State:</b>High</p>

Port Name	I/O	Description
sda_in_a	I	<p>Incoming SDA Input SDA Signal from the SDA Pad Driver.</p> <p><b>Exists:</b>Always</p> <p><b>Synchronous To:</b>(IC_DEVICE_ROLE==4) &amp;&amp; (IC_SPEED_HDR_TS==1) ? "hdr_tx_clk,pclk,scl_in_a,scl_in_a_n" : ((IC_DEVICE_ROLE==4) &amp;&amp; (IC_SPEED_HDR_TS==0)) ? "pclk,scl_in_a,scl_in_a_n" : "None"</p> <p><b>Registered:</b>No</p> <p><b>Power Domain:</b>SINGLE_DOMAIN</p> <p><b>Active State:</b>High</p>

### 4.3 Slave Interface Signals

rst_n	-ccc_type
mode_i2c	-ccc_def_byte
slv_hj_ctrl	-ccc_vld_t
scan_clk	-ccc_addmtch_dir_t
slv_perr_ack_onc	-ccc_wr_data_vld_t
slv_par_err_dis	-ccc_ongoing
slv_par_err_ext	-ccc_wr_data
ccc_wr_data_rdy	-ccc_rd_data_rdy_t
ccc_rd_data_vld	-slv_pec_error
ccc_rd_data	-i2c_glitch_filter_en
slv_pec_enable	-cmd_code
slv_pec_err_ext	-sub_addr
sub_addr_size	-sub_addr_vld_t
wr_data_rdy	-wr_data_vld_t
rd_data_vld	-wr_data
rd_data	-wr_par_err_t
hj_req_now	-wr_crc_err_t
sir_req_on_start	-slv_frame_err_t
sir_req_now	-rd_data_rdy_t
sir_data	-sir_data_ongoing
sir_data_vld	-sir_data_rdy_t
sir_mdb	-ibi_done_t
act_mode	-ibi_sts
pending_int	-act_state
slv_s1_err_rst	-slv_lite_wr_ongoing
slv_s0_disable	-slv_lite_rd_ongoing
static_addr_en	-mrl
static_addr	-mwl
inst_id	-ibi_payload_size
slv_pid	-bus_idle
slv_dcr	-slv_dynamic_addr_vld
	-slv_en_hj
	-slv_en_int
	-wakeup

**Table 4-3 Slave Interface Signals**

Port Name	I/O	Description
rst_n	I	Slave Lite Interface Reset  <b>Exists:</b> (IC_DEVICE_ROLE==5) <b>Synchronous To:</b> None <b>Registered:</b> N/A <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> Low

Port Name	I/O	Description
mode_i2c	I	<p>I2C or I3C mode select signal Applicable only in Slave mode of operation. <b>Exists:</b>(IC_DEVICE_ROLE&gt;1) <b>Synchronous To:</b>None <b>Registered:</b>No <b>Power Domain:</b>SINGLE_DOMAIN <b>Active State:</b>Not applicable</p>
slv_hj_ctrl	I	<p>Control to disable Slave Hot-Join feature This port should be tied to 1 to enable Slave Hot-join feature. If this port is tied to 0, the controller behaves as a non Hot-join device, even though Hot-Join capability is selected (IC_SLV_HJ to 1). The slave application is expected to drive hj_req_now port to 0 if slv_hj_ctrl port is tied to 0. <b>Exists:</b>(IC_DEVICE_ROLE==5) &amp;&amp; (IC_SLV_HJ==1) <b>Synchronous To:</b>scl_in_a <b>Registered:</b>No <b>Power Domain:</b>SINGLE_DOMAIN <b>Active State:</b>Not applicable</p>
scan_clk	I	<p>Slave Test Mode Clock <b>Exists:</b>(IC_DEVICE_ROLE==5) <b>Synchronous To:</b>None <b>Registered:</b>No <b>Power Domain:</b>SINGLE_DOMAIN <b>Active State:</b>Low</p>
slv_perr_ack_anc	I	<p>ack after completion of transfer with parity error <b>Exists:</b>(IC_DEVICE_ROLE==5) <b>Synchronous To:</b>scl_in_a,scl_in_a_n <b>Registered:</b>No <b>Power Domain:</b>SINGLE_DOMAIN <b>Active State:</b>High</p>
slv_par_err_dis	I	<p>disable parity error <b>Exists:</b>(IC_DEVICE_ROLE==5) <b>Synchronous To:</b>scl_in_a <b>Registered:</b>No <b>Power Domain:</b>SINGLE_DOMAIN <b>Active State:</b>High</p>



Port Name	I/O	Description
slv_par_err_ext	I	external parity error  <b>Exists:</b> (IC_DEVICE_ROLE==5) <b>Synchronous To:</b> scl_in_a <b>Registered:</b> No <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High
ccc_type[7:0]	O	ccc type  <b>Exists:</b> (IC_DEVICE_ROLE==5) <b>Synchronous To:</b> scl_in_a <b>Registered:</b> Yes <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High
ccc_def_byte[7:0]	O	ccc def byte  <b>Exists:</b> (IC_DEVICE_ROLE==5) <b>Synchronous To:</b> scl_in_a <b>Registered:</b> Yes <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High
ccc_vld_t	O	ccc vld toggle  <b>Exists:</b> (IC_DEVICE_ROLE==5) <b>Synchronous To:</b> scl_in_a <b>Registered:</b> Yes <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High
ccc_addmtch_dir_t	O	ccc addrmatch for dir ccc toggle  <b>Exists:</b> (IC_DEVICE_ROLE==5) <b>Synchronous To:</b> scl_in_a <b>Registered:</b> Yes <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High
ccc_wr_data_vld_t	O	ccc wr data valid toggle  <b>Exists:</b> (IC_DEVICE_ROLE==5) <b>Synchronous To:</b> scl_in_a <b>Registered:</b> Yes <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High

Port Name	I/O	Description
ccc_ongoing	O	ccc ongoing  <b>Exists:</b> (IC_DEVICE_ROLE==5) <b>Synchronous To:</b> scl_in_a_n <b>Registered:</b> Yes <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High
ccc_wr_data[7:0]	O	ccc_wr_data  <b>Exists:</b> (IC_DEVICE_ROLE==5) <b>Synchronous To:</b> scl_in_a <b>Registered:</b> Yes <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High
ccc_wr_data_rdy	I	ccc wr data ready  <b>Exists:</b> (IC_DEVICE_ROLE==5) <b>Synchronous To:</b> scl_in_a,scl_in_a_n <b>Registered:</b> No <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High
ccc_rd_data_vld	I	ccc rd data valid  <b>Exists:</b> (IC_DEVICE_ROLE==5) <b>Synchronous To:</b> scl_in_a,scl_in_a_n <b>Registered:</b> No <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High
ccc_rd_data[7:0]	I	ccc_rd_data  <b>Exists:</b> (IC_DEVICE_ROLE==5) <b>Synchronous To:</b> scl_in_a <b>Registered:</b> No <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High
ccc_rd_data_rdy_t	O	ccc rd data ready toggle  <b>Exists:</b> (IC_DEVICE_ROLE==5) <b>Synchronous To:</b> scl_in_a <b>Registered:</b> Yes <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High

Port Name	I/O	Description
slv_pec_enable	I	<p>slv_pec_enable The input from application to enable Packet Error Check The application enables the PEC based on PEC enable from Master</p> <p><b>Exists:</b>(IC_DEVICE_ROLE==5)&amp;&amp;(IC_HAS_PEC==1)  <b>Synchronous To:</b>scl_in_a  <b>Registered:</b>No  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>
slv_pec_err_ext	I	<p>pec error from application register The input from application will keep asserting the status of PEC error after the GETSTATUS CCC has been received.</p> <p><b>Exists:</b>(IC_DEVICE_ROLE==5)&amp;&amp;(IC_HAS_PEC==1)  <b>Synchronous To:</b>scl_in_a  <b>Registered:</b>No  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>
slv_pec_error	O	<p>slv_pec_error Indicates the controller has detected a PEC error</p> <p><b>Exists:</b>(IC_DEVICE_ROLE==5)&amp;&amp;(IC_HAS_PEC==1)  <b>Synchronous To:</b>scl_in_a  <b>Registered:</b>No  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>
sub_addr_size[1:0]	I	<p>Sub-Address size</p> <p><b>Exists:</b>(IC_DEVICE_ROLE==5) &amp;&amp; (IC_HAS_SUB_ADDRESS==1)  <b>Synchronous To:</b>scl_in_a  <b>Registered:</b>No  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>
wr_data_rdy	I	<p>Write Data Ready Applicable only in Slave mode of operation.</p> <p><b>Exists:</b>(IC_DEVICE_ROLE==5)  <b>Synchronous To:</b>(IC_DEVICE_ROLE==5) ? "scl_in_a,scl_in_a_n" : "scl_in_a"  <b>Registered:</b>No  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>

Port Name	I/O	Description
rd_data_vld	I	<p>Read Data Valid Applicable only in Slave mode of operation.</p> <p><b>Exists:</b>(IC_DEVICE_ROLE==5)  <b>Synchronous To:</b>scl_in_a,scl_in_a_n  <b>Registered:</b>No  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>
rd_data[(IC_SLV_READ_DW-1):0]	I	<p>Read data Applicable only in Slave mode of operation.</p> <p><b>Exists:</b>(IC_DEVICE_ROLE==5)  <b>Synchronous To:</b>scl_in_a  <b>Registered:</b>No  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>
hj_req_now	I	<p>Genrate Hot-Join request now</p> <p><b>Exists:</b>(IC_DEVICE_ROLE==5) &amp;&amp; (IC_SLV_HJ==1)  <b>Synchronous To:</b>scl_in_a  <b>Registered:</b>No  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>
sir_req_on_start	I	<p>Generate SIR request on START</p> <p><b>Exists:</b>(IC_DEVICE_ROLE==5) &amp;&amp; (IC_SLV_HAS_IBI==1)  <b>Synchronous To:</b>scl_in_a_n  <b>Registered:</b>(IC_SLV_IBI==1) ? "Yes" : "No"  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>
sir_req_now	I	<p>Generate SIR request now</p> <p><b>Exists:</b>(IC_DEVICE_ROLE==5) &amp;&amp; (IC_SLV_HAS_IBI==1)  <b>Synchronous To:</b>scl_in_a,scl_in_a_n  <b>Registered:</b>No  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>

Port Name	I/O	Description
sir_data[7:0]	I	<p>Data along with Slave Interrupt Request</p> <p><b>Exists:</b>(IC_DEVICE_ROLE==5) &amp;&amp; (IC_SLV_IBI_DATA==1)  <b>Synchronous To:</b>scl_in_a  <b>Registered:</b>No  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>
sir_data_vld	I	<p>SIR data valid</p> <p><b>Exists:</b>(IC_DEVICE_ROLE==5) &amp;&amp; (IC_SLV_IBI_DATA==1)  <b>Synchronous To:</b>scl_in_a,scl_in_a_n  <b>Registered:</b>No  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>
sir_mdb[7:0]	I	<p>SIR mandatory data byte</p> <p><b>Exists:</b>(IC_DEVICE_ROLE==5) &amp;&amp; (IC_SLV_IBI_DATA==1)  <b>Synchronous To:</b>scl_in_a  <b>Registered:</b>No  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>
act_mode[1:0]	I	<p>Slave activity mode for GETSTATUS CCC  Applicable only in Slave mode of operation.</p> <p><b>Exists:</b>(IC_DEVICE_ROLE&gt;1)  <b>Synchronous To:</b>(IC_SLAVE_APB_EN==1) &amp;&amp;  (IC_MAIN_MASTER_EN==0) ? "pclk" :  (IC_SLAVE_LITE_ONLY_EN==1) ? "scl_in_a": "core_clk"  <b>Registered:</b>(IC_DEVICE_ROLE==5) ? "Yes" : "No"  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>
pending_int[3:0]	I	<p>Pending interrupt information for GETSTATUS CCC  Applicable only in Slave mode of operation.</p> <p><b>Exists:</b>(IC_DEVICE_ROLE&gt;1)  <b>Synchronous To:</b>(IC_SLAVE_APB_EN==1) &amp;&amp;  (IC_MAIN_MASTER_EN==0) ? "pclk" :  (IC_SLAVE_LITE_ONLY_EN==1) ? "scl_in_a": "core_clk"  <b>Registered:</b>(IC_DEVICE_ROLE==5) ? "Yes" : "No"  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>

Port Name	I/O	Description
slv_s1_err_rst	I	<p>slave S1 error reset Applicable in only Slave mode of operation. This signal if asserted will enable the controller to detect start and stop after S1 error has been asserted without an hdr exit pattern detection. This is feature is for future version of the IP. It will be used for recovering from S1 error when the bus is idle for 1 micro sec. This should be tied to a constant value of 1 for this release.</p> <p><b>Exists:</b>(IC_DEVICE_ROLE==5)  <b>Synchronous To:</b>sda_in_a,sda_in_a_n  <b>Registered:</b>No  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>
slv_s0_disable	I	<p>slave S0 error disable Applicable in only Slave mode of operation. This signal if asserted will disable the controller to detect S0 error.</p> <p><b>Exists:</b>(IC_DEVICE_ROLE==5)  <b>Synchronous To:</b>scl_in_a_n  <b>Registered:</b>No  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>
static_addr_en	I	<p>Slave static address valid Applicable only in Slave mode of operation.</p> <p><b>Exists:</b>(IC_DEVICE_ROLE&gt;1)  <b>Synchronous To:</b>None  <b>Registered:</b>No  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>
static_addr[6:0]	I	<p>Slave static address Applicable only in Slave mode of operation.</p> <p><b>Exists:</b>(IC_DEVICE_ROLE&gt;1)  <b>Synchronous To:</b>None  <b>Registered:</b>No  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>
inst_id[3:0]	I	<p>Slave instance ID Applicable only in Slave mode of operation.</p> <p><b>Exists:</b>(IC_DEVICE_ROLE&gt;1) &amp;&amp;  (IC_SLV_UNIQUE_ID_PROG==0)  <b>Synchronous To:</b>None  <b>Registered:</b>No  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>

Port Name	I/O	Description
i2c_glitch_filter_en	O	<p>I2C 50ns glitch filter enable Applicable only in Slave mode of operation.</p> <p><b>Exists:</b>(IC_DEVICE_ROLE&gt;1)  <b>Synchronous To:</b>(IC_SLAVE_APB_EN==1) &amp;&amp;  (IC_MAIN_MASTER_EN==0) ? "pclk" : (IC_DEVICE_ROLE==5) ?  "scl_in_a" : "core_clk"  <b>Registered:</b>Yes  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>
cmd_code[7:0]	O	<p>Slave HDR command code Applicable only in Slave mode of operation.</p> <p><b>Exists:</b>(IC_DEVICE_ROLE==5) &amp;&amp; ((IC_SPEED_HDR_TS==1)     (IC_SPEED_HDR_DDR==1))  <b>Synchronous To:</b>scl_in_a  <b>Registered:</b>Yes  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>
sub_addr[31:0]	O	<p>Slave sub address</p> <p><b>Exists:</b>(IC_DEVICE_ROLE==5) &amp;&amp; (IC_HAS_SUB_ADDRESS==1)  <b>Synchronous To:</b>scl_in_a  <b>Registered:</b>Yes  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>
sub_addr_vld_t	O	<p>Slave sub address valid toggle</p> <p><b>Exists:</b>(IC_DEVICE_ROLE==5) &amp;&amp; (IC_HAS_SUB_ADDRESS==1)  <b>Synchronous To:</b>scl_in_a  <b>Registered:</b>Yes  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>Toggle</p>
wr_data_vld_t	O	<p>Slave write data valid toggle Applicable only in Slave mode of operation.</p> <p><b>Exists:</b>(IC_DEVICE_ROLE==5)  <b>Synchronous To:</b>scl_in_a_n  <b>Registered:</b>Yes  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>Toggle</p>

Port Name	I/O	Description
wr_data[(IC_RXC_HDR_DL_2_TL_DW-1):0]	O	<p>Slave write data toggle Applicable only in Slave mode of operation.</p> <p><b>Exists:</b>(IC_DEVICE_ROLE==5)  <b>Synchronous To:</b>scl_in_a_n  <b>Registered:</b>Yes  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>
wr_par_err_t	O	<p>Slave write data parity error</p> <p><b>Exists:</b>(IC_DEVICE_ROLE==5)  <b>Synchronous To:</b>scl_in_a_n  <b>Registered:</b>Yes  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>
wr_crc_err_t	O	<p>Slave write data CRC error</p> <p><b>Exists:</b>(IC_DEVICE_ROLE==5) &amp;&amp; (IC_SPEED_HDR_DDR==1)  <b>Synchronous To:</b>scl_in_a  <b>Registered:</b>Yes  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>
slv_frame_err_t	O	<p>Slave frame error</p> <p><b>Exists:</b>(IC_DEVICE_ROLE==5) &amp;&amp; (IC_SPEED_HDR_DDR==1)  <b>Synchronous To:</b>scl_in_a_n  <b>Registered:</b>Yes  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>High</p>
rd_data_rdy_t	O	<p>Slave read data ready toggle</p> <p><b>Exists:</b>(IC_DEVICE_ROLE==5)  <b>Synchronous To:</b>scl_in_a  <b>Registered:</b>Yes  <b>Power Domain:</b>SINGLE_DOMAIN  <b>Active State:</b>Toggle</p>



Port Name	I/O	Description
sir_data_ongoing	O	Slave SIR read data ongoing  <b>Exists:</b> (IC_DEVICE_ROLE==5) && (IC_SLV_IBI_DATA==1) <b>Synchronous To:</b> scl_in_a <b>Registered:</b> Yes <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High
sir_data_rdy_t	O	Slave SIR read data ready toggle  <b>Exists:</b> (IC_DEVICE_ROLE==5) && (IC_SLV_IBI_DATA==1) <b>Synchronous To:</b> scl_in_a <b>Registered:</b> Yes <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> Toggle
ibi_done_t	O	Slave SIR done toggle  <b>Exists:</b> (IC_DEVICE_ROLE==5) && (IC_SLV_HJ==1    IC_SLV_IBI==1) <b>Synchronous To:</b> scl_in_a <b>Registered:</b> Yes <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> Toggle
ibi_sts[1:0]	O	Slave IBI status  <b>Exists:</b> (IC_DEVICE_ROLE==5) && (IC_SLV_HJ==1    IC_SLV_IBI==1) <b>Synchronous To:</b> scl_in_a <b>Registered:</b> Yes <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High
act_state[1:0]	O	Slave activity state Applicable only in Slave mode of operation.  <b>Exists:</b> (IC_DEVICE_ROLE==5) <b>Synchronous To:</b> scl_in_a <b>Registered:</b> Yes <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High

Port Name	I/O	Description
slv_lite_wr_ongoing	O	Slave write ongoing status  <b>Exists:</b> (IC_DEVICE_ROLE==5) <b>Synchronous To:</b> scl_in_a <b>Registered:</b> Yes <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High
slv_lite_rd_ongoing	O	Slave read ongoing status  <b>Exists:</b> (IC_DEVICE_ROLE==5) <b>Synchronous To:</b> scl_in_a <b>Registered:</b> Yes <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High
mrl[15:0]	O	Slave maximum read length  <b>Exists:</b> (IC_DEVICE_ROLE==5) <b>Synchronous To:</b> scl_in_a <b>Registered:</b> Yes <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High
mwl[15:0]	O	Slave maximum write length  <b>Exists:</b> (IC_DEVICE_ROLE==5) <b>Synchronous To:</b> scl_in_a <b>Registered:</b> Yes <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High
ibi_payload_size[7:0]	O	Slave maximum IBI payload size  <b>Exists:</b> (IC_DEVICE_ROLE==5) &&(IC_SLV_IBI_DATA==1) <b>Synchronous To:</b> scl_in_a <b>Registered:</b> Yes <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High
bus_idle	O	Slave bus idle  <b>Exists:</b> (IC_DEVICE_ROLE==5) <b>Synchronous To:</b> scl_in_a_n <b>Registered:</b> Yes <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High

Port Name	I/O	Description
slv_dynamic_addr_vld	O	Slave dynamic address valid  <b>Exists:</b> (IC_DEVICE_ROLE==5) <b>Synchronous To:</b> scl_in_a <b>Registered:</b> Yes <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High
slv_en_hj	O	Slave Hot-Join Event Enable  <b>Exists:</b> (IC_DEVICE_ROLE==5) && (IC_SLV_HJ==1) <b>Synchronous To:</b> scl_in_a_n <b>Registered:</b> Yes <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High
slv_en_int	O	Slave Interrupt Request Event Enable  <b>Exists:</b> (IC_DEVICE_ROLE==5) && (IC_SLV_IBI==1) <b>Synchronous To:</b> scl_in_a_n <b>Registered:</b> Yes <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High
wakeup	O	Slave wakeup signal Applicable only in Slave mode of operation. <b>Exists:</b> IC_DEVICE_ROLE>1 <b>Synchronous To:</b> scl_in_a <b>Registered:</b> (IC_SPEED_HDR_TS_EN==1) ? "No" : "Yes" <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High
slv_pid[47:0]	I	Slave Device 48-bit Provisioned ID Applicable in only Slave mode of operation.  <b>Exists:</b> (IC_DEVICE_ROLE>1) && (IC_SLV_UNIQUE_ID_PROG==1) <b>Synchronous To:</b> None <b>Registered:</b> No <b>Power Domain:</b> SINGLE_DOMAIN <b>Active State:</b> High

Port Name	I/O	Description
slv_dcr[7:0]	I	<p>Device Characteristic Register value Applicable in only Slave mode of operation.</p> <p><b>Exists:</b>(IC_DEVICE_ROLE&gt;1) &amp;&amp; (IC_SLV_UNIQUE_ID_PROG==1) <b>Synchronous To:</b>None <b>Registered:</b>No <b>Power Domain:</b>SINGLE_DOMAIN <b>Active State:</b>High</p>

## 4.4 Scan Interface Signals

slv\_test\_mode - 

**Table 4-4** Scan Interface Signals

Port Name	I/O	Description
slv_test_mode	I	<p>Test Mode</p> <p>Used to ensure that test automation tools can control all asynchronous flip-flop signals. During scan, set this signal high all the time. In normal operation, tie this signal low.</p> <p><b>Exists:</b> (IC_DEVICE_ROLE &gt; 1)    ((IC_DEVICE_ROLE == 1) &amp;&amp; (IC_SPEED_HDR_DDR == 1) &amp;&amp; (IC_SDA_SAMPLING_IN_SCL == 1))</p> <p><b>Synchronous To:</b> None</p> <p><b>Registered:</b> No</p> <p><b>Power Domain:</b> SINGLE_DOMAIN</p> <p><b>Active State:</b> Low</p>

Synopsys confidential document, provided to GigaDevice  
Semiconductor under NDA, do not redistribute

# A

## Area

---

This appendix provides area in terms of gate count.

Synopsys confidential document, provided to GigaDevice  
Semiconductor under NDA, do not redistribute

## A.1 Area

The following table provides the area in terms of gate count for the various configurations.

**Note**

Industry Standard 28nm synthesis library is used to extract the following Area information.

**Table A-1 Gate Count**

Product	Configuration	Role	Gate Count
DWC_mipi_i3c	Slave-Lite Min Configuration	Tx side bus width = 8, No HJ, No IBI, No DDR	2.5 K Gates
DWC_mipi_i3c	Slave-Lite Max Configuration	Tx side bus width = 32, HJ selected, IBI selected, DDR selected	4.2 K Gates



# B

## Synchronizer Methods

---

---

### Note

- Slave-Lite does not use any synchronizer methods.
  - All the signals must be synchronized and given to the Slave-Lite.
-

Synopsys confidential document, provided to GigaDevice  
Semiconductor under NDA, do not redistribute

## CCC Types which Use CCC Channel

Table C-1 CCC which use the CCC channel of the Controller

CCC Type	B/D	CCC Channel Used	Functionality Supported by Slave-Lite	Remarks
ENEC	B/D	No	Yes	
DISEC	B/D	No	Yes	
ENTAS0	B/D	No	Yes	
ENTAS1	B/D	No	Yes	
ENTAS2	B/D	No	Yes	
ENTAS3	B/D	No	Yes	
RSTDAA	B	Yes	Yes	
ENTDAA	B	No	Yes	
DEFSLVS	B	Yes	No	This is specific to secondary master in Slave-Lite mode. This comes out of the channel as it is not internally consumed.
SETMWL	B/D	No	Yes	
SETMRL	B/D	No	Yes	
ENTMM	B	Yes	No	This ccc is not consumed internally by the Slave-Lite IP and hence it comes out of the CCC channel.
SETBUSCON	B	Yes	No	This ccc is not consumed internally by the Slave-Lite IP and hence it comes out of the CCC channel.
ENDXFER	B/D	Yes	No	This ccc is not consumed internally by the Slave-Lite IP and hence it comes out of the CCC channel.
ENTHDR0	B	No	Yes	
ENTHDR1	B	No	Yes	

CCC Type	B/D	CCC Channel Used	Functionality Supported by Slave-Lite	Remarks
ENTHDR2	B	No	Yes	
ENTHDR3	B	Yes	Yes	
ENTHDR4	B	Yes	No	This ccc is not consumed internally by the Slave-Lite IP and hence it comes out of the CCC channel.
ENTHDR5	B	Yes	No	This ccc is not consumed internally by the Slave-Lite IP and hence it comes out of the CCC channel.
ENTHDR6	B	Yes	No	This ccc is not consumed internally by the Slave-Lite IP and hence it comes out of the CCC channel.
ENTHDR7	B	Yes	No	This ccc is not consumed internally by the Slave-Lite IP and hence it comes out of the CCC channel.
SETXTIME	B/D	No	Yes	This functionality is supported only in time stamping mode. In direct mode can be nacked by making ccc_wr_data_vld low
SETAASA	B	Yes	Yes	
RSTACT	B/D	Yes	Yes	
DEFGRPA	B	Yes	No	This ccc is not consumed internally by the Slave-Lite IP and hence it comes out of the CCC channel.
RSTGRPA	B	Yes	No	This ccc is not consumed internally by the Slave-Lite IP and hence it comes out of the CCC channel.
MLANE	B	Yes	No	This ccc is not consumed internally by the Slave-Lite IP and hence it comes out of the CCC channel.
SETHID	B	Yes	Yes	
DEVCTRL	B	Yes	Yes	
SETDASA	D	No	Yes	
SETNEWDA	D	No	Yes	
GETMWL	D	No	Yes	
GETMRL	D	No	Yes	
GETPID	D	No	Yes	
GETBCR	D	No	Yes	
GETDCR	D	No	Yes	
GETSTATUS	D	No	Yes	

CCC Type	B/D	CCC Channel Used	Functionality Supported by Slave-Lite	Remarks
GETACCMST	D	Yes	Yes	This is specific to secondary master in Slave-Lite mode. This comes out of the channel as it is not internally consumed.
SETBRGTGT	D	Yes	No	This ccc is not consumed internally by the Slave-Lite IP and hence it comes out of the CCC channel.
GETMXDS	D	No	Yes	
GETCAPS	D	No	Yes	
SETRROUTE	D	Yes	No	This ccc is not consumed internally by the Slave-Lite IP and hence it comes out of the CCC channel.
D2DXFER	D	Yes	No	This ccc is not consumed internally by the Slave-Lite IP and hence it comes out of the CCC channel.
GETXTIME	D	No	Yes	
DEVCAP	D	Yes	Yes	

**Note**

Directed RSTDAA that is deprecated in the latest I3C specification is NACK'd and is not passed through the channel.

Synopsys confidential document, provided to GigaDevice  
Semiconductor under NDA, do not redistribute