



# Specification for I3C Host Controller Interface (I3C HCI<sup>SM</sup>)

**Version 1.2**

**15 February 2023**

MIPI Board Adopted 12 April 2023

**Public Release Edition**

Further technical changes to this document are expected as work continues in the Software Working Group.

**NOTICE OF DISCLAIMER**

The material contained herein is provided on an “AS IS” basis. To the maximum extent permitted by applicable law, this material is provided AS IS AND WITH ALL FAULTS, and the authors and developers of this material and MIPI Alliance Inc. (“MIPI”) hereby disclaim all other warranties and conditions, either express, implied or statutory, including, but not limited to, any (if any) implied warranties, duties or conditions of merchantability, of fitness for a particular purpose, of accuracy or completeness of responses, of results, of workmanlike effort, of lack of viruses, and of lack of negligence. ALSO, THERE IS NO WARRANTY OR CONDITION OF TITLE, QUIET ENJOYMENT, QUIET POSSESSION, CORRESPONDENCE TO DESCRIPTION OR NON-INFRINGEMENT WITH REGARD TO THIS MATERIAL.

IN NO EVENT WILL ANY AUTHOR OR DEVELOPER OF THIS MATERIAL OR MIPI BE LIABLE TO ANY OTHER PARTY FOR THE COST OF PROCURING SUBSTITUTE GOODS OR SERVICES, LOST PROFITS, LOSS OF USE, LOSS OF DATA, OR ANY INCIDENTAL, CONSEQUENTIAL, DIRECT, INDIRECT, OR SPECIAL DAMAGES WHETHER UNDER CONTRACT, TORT, WARRANTY, OR OTHERWISE, ARISING IN ANY WAY OUT OF THIS OR ANY OTHER AGREEMENT RELATING TO THIS MATERIAL, WHETHER OR NOT SUCH PARTY HAD ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGES.

The material contained herein is not a license, either expressly or impliedly, to any IPR owned or controlled by any of the authors or developers of this material or MIPI. Any license to use this material is granted separately from this document. This material is protected by copyright laws, and may not be reproduced, republished, distributed, transmitted, displayed, broadcast or otherwise exploited in any manner without the express prior written permission of MIPI Alliance. MIPI, MIPI Alliance and the dotted rainbow arch and all related trademarks, service marks, tradenames, and other intellectual property are the exclusive property of MIPI Alliance Inc. and cannot be used without its express prior written permission. The use or implementation of this material may involve or require the use of intellectual property rights (“IPR”) including (but not limited to) patents, patent applications, or copyrights owned by one or more parties, whether or not members of MIPI. MIPI does not make any search or investigation for IPR, nor does MIPI require or request the disclosure of any IPR or claims of IPR as respects the contents of this material or otherwise.

Without limiting the generality of the disclaimers stated above, users of this material are further notified that MIPI: (a) does not evaluate, test or verify the accuracy, soundness or credibility of the contents of this material; (b) does not monitor or enforce compliance with the contents of this material; and (c) does not certify, test, or in any manner investigate products or services or any claims of compliance with MIPI specifications or related material.

Questions pertaining to this material, or the terms or conditions of its provision, should be addressed to:

MIPI Alliance, Inc.  
c/o IEEE-ISTO  
445 Hoes Lane, Piscataway New Jersey 08854, United States  
Attn: Executive Director

# Contents

<b>Figures .....</b>	<b>vii</b>
<b>Tables .....</b>	<b>ix</b>
<b>Release History .....</b>	<b>xiii</b>
<b>1    Introduction .....</b>	<b>1</b>
1.1    Scope .....	1
1.2    Purpose .....	1
<b>2    Terminology .....</b>	<b>2</b>
2.1    Use of Special Terms .....	2
2.2    Definitions .....	2
2.3    Abbreviations .....	7
2.4    Acronyms .....	7
2.5    Color Coding .....	9
<b>3    References .....</b>	<b>10</b>
<b>4    Technical Overview .....</b>	<b>11</b>
4.1    Scope .....	12
4.2    I3C HCI Purpose .....	12
4.3    I3C HCI Key Features .....	13
4.4    I3C HCI Fundamental Principles .....	14
4.5    I3C HCI Relationship to Other MIPI Specifications .....	15
4.6    What's New in I3C HCI Version 1.2 .....	16
<b>5    Architectural Overview (informative).....</b>	<b>19</b>
5.1    Interface Architecture .....	19
5.2    General Information .....	21
5.3    Command Interface .....	23
5.4    Rings Overview (DMA Mode).....	26
5.5    Extended Capabilities .....	29
5.6    Target Device Support Model.....	29
5.6.1    I3C Devices .....	29
5.6.2    I <sup>2</sup> C Devices .....	29
<b>6    Theory of Operation.....</b>	<b>31</b>
6.1    Host Controller Management .....	32
6.1.1    Host Controller Initialization .....	32
6.1.2    Host Controller De-Initialization .....	34
6.1.3    Host Controller Reset .....	34
6.1.4    Power Management .....	35
6.2    Host System Memory Access .....	37
6.2.1    Scatter-Gather for Host System Memory .....	38
6.3    Device Management .....	41
6.3.1    Device Attach, Enumeration, and Initialization .....	41
6.3.2    Device Detach, Reset, and Power Management .....	45
6.3.3    Device Context .....	46
6.4    Device Addressing .....	48
6.4.1    Dynamic Address Assignment with ENTDAA .....	49
6.4.2    Using Static Addresses .....	50
6.4.3    Grouped Addressing .....	51
6.5    PIO Queue Management .....	52
6.5.1    PIO Section Initialization .....	55
6.5.2    Command Queue Operation .....	56
6.5.3    Response Queue Operation .....	58
6.5.4    IBI Queue Operation .....	60

6.5.5	PIO Thresholds.....	63
6.5.6	PIO Abort Operation .....	66
6.6	Ring Management .....	67
6.6.1	Ring Bundle Initialization .....	67
6.6.2	Command / Response Ring Pairs .....	70
6.6.3	IBI Ring Pairs.....	75
6.6.4	Ring Header Registers.....	81
6.6.5	Ring Bundle Servicing .....	83
6.6.6	Ring Abort Operation .....	85
6.7	Transfer Command Handling .....	87
6.8	Transfers .....	88
6.8.1	Transfers in PIO Mode .....	88
6.8.2	Transfers in DMA Mode .....	90
6.8.3	Data Byte Ordering .....	94
6.8.4	Host Controller Abort Operation .....	96
6.8.5	Support for I3C ‘Short’ Read-Type Transfers.....	99
6.8.6	Support for Transfer Termination and HDR Mode Configuration.....	99
6.9	IBI Handling.....	101
6.9.1	IBI Handling in PIO Mode.....	101
6.9.2	IBI Handling in DMA Mode .....	102
6.9.3	Timestamping.....	104
6.9.4	IBI Data Abort Operation .....	105
6.9.5	Target IBI Credit Counting.....	106
6.10	Managed CCC Transfer Framing Model .....	109
6.10.1	Direct CCCs and NACK Retry.....	109
6.10.2	Providing Data for Broadcast/Direct Write CCCs.....	109
6.10.3	Error Handling for CCC Flows .....	110
6.11	Auto-Command .....	111
6.11.1	Flow of Operation .....	111
6.11.2	Configuration with DAT .....	112
6.11.3	Data Reporting to IBI Payload Buffer .....	112
6.12	Request Handling .....	114
6.12.1	PIO Mode .....	116
6.12.2	DMA Mode .....	117
6.13	Error Handling.....	118
6.13.1	Error Status Codes in Response Descriptor.....	119
6.13.2	Errors Due to Command Sequence Stall or Timeout.....	121
6.14	Interrupts .....	123
6.15	Target Reset and Bus Recovery .....	125
6.15.1	Target Reset Pattern.....	125
6.15.2	Controller SDA Recovery or Bus Reset Procedures.....	130
6.16	Scheduled Commands .....	132
6.16.1	Common Aspects of Command Scheduling.....	134
6.16.2	Handler Type 1: Simple Table .....	138
6.16.3	Handler Type 2: Command Sequence .....	140
6.16.4	Handler Type 3: Schedule Buffer .....	142
6.17	Standby Controller Mode .....	144
6.17.1	Configuration and Modes of Operation.....	150
6.17.2	Transition from Active Controller Mode to Standby Controller Mode.....	156
6.17.3	Operation in Standby Controller Mode .....	165
6.17.4	Transition from Standby Controller Mode to Active Controller Mode.....	173
6.18	Dead Bus Recovery .....	181

<b>7 Register Interface .....</b>	<b>183</b>
7.1 Register Memory Access Conventions .....	183
7.2 PCI Configuration Registers.....	183
7.3 Register Map .....	184
7.3.1 Capabilities and Operation Registers.....	186
7.3.2 Extended Capabilities Registers.....	188
7.3.3 DAT and DCT Registers.....	189
7.3.4 Ring Headers Registers .....	191
7.3.5 Programmable I/O Registers .....	192
7.4 Host Controller Capability & Operations Registers .....	193
7.4.1 HCI Version (HCI_VERSION) (BASE + 0x0) .....	193
7.4.2 Host Controller Control (HC_CONTROL) (BASE + 0x4) .....	194
7.4.3 Controller Device Address (CONTROLLER_DEVICE_ADDR) (BASE + 0x8) .....	199
7.4.4 Host Controller Capabilities (HC_CAPABILITIES) (BASE + 0xC) .....	201
7.4.5 Reset Control (RESET_CONTROL) (BASE + 0x10).....	204
7.4.6 Present State (PRESENT_STATE) (BASE + 0x14).....	206
7.4.7 Interrupt Status (INTR_STATUS) (BASE + 0x20) .....	207
7.4.8 Interrupt Status Enable (INTR_STATUS_ENABLE) (BASE + 0x24).....	208
7.4.9 Interrupt Signal Enable (INTR_SIGNAL_ENABLE) (BASE + 0x28).....	209
7.4.10 Interrupt Force (INTR_FORCE) (BASE + 0x2C).....	210
7.4.11 Device Address Table Section Offset (DAT_SECTION_OFFSET) (BASE + 0x30) .....	211
7.4.12 Device Characteristics Table Section Offset (DCT_SECTION_OFFSET) (BASE + 0x34) .....	212
7.4.13 Ring Headers Section Offset (RING_HEADERS_SECTION_OFFSET) (BASE + 0x38) .....	213
7.4.14 PIO Section Offset (PIO_SECTION_OFFSET) (BASE + 0x3C) .....	213
7.4.15 Extended Capabilities Section Offset (EXT_CAPS_SECTION_OFFSET) (BASE + 0x40) .....	214
7.4.16 Internal Control Command Subtype Support (INT_CTRL_CMDS_EN) (BASE + 0x4C) .....	215
7.4.17 IBI Notify Control (IBI_NOTIFY_CTRL) (BASE + 0x58).....	216
7.4.18 IBI Data Abort Control (IBI_DATA_ABORT_CTRL) (BASE + 0x5C) .....	218
7.4.19 Device Context Base Address Low (DEV_CTX_BASE_LO) (BASE + 0x60) .....	220
7.4.20 Device Context Base Address High (DEV_CTX_BASE_HI) (BASE + 0x64).....	221
7.4.21 Device Context Scatter-Gather Support (DEV_CTX_SG) (BASE + 0x68) .....	221
7.5 PIO Mode Registers .....	222
7.5.1 Command Queue Port (COMMAND_QUEUE_PORT) (PIO + 0x0) .....	222
7.5.2 Response Queue Port (RESPONSE_QUEUE_PORT) (PIO + 0x4).....	222
7.5.3 Transfer Data Port (XFER_DATA_PORT) (PIO + 0x8) .....	223
7.5.4 IBI Port (IBI_PORT) (PIO + 0xC) .....	224
7.5.5 Queue Threshold Control (QUEUE_THLD_CTRL) (PIO + 0x10).....	225
7.5.6 Transfer Data Buffer Threshold Control (DATA_BUFFER_THLD_CTRL) (PIO + 0x14) .....	227
7.5.7 Queue Size (QUEUE_SIZE) (PIO + 0x18) .....	229
7.5.8 Alternate Queue Size (ALT_QUEUE_SIZE) (PIO + 0x1C) .....	231
7.5.9 PIO Interrupt Status (PIO_INTR_STATUS) (PIO + 0x20) .....	233
7.5.10 PIO Interrupt Status Enable (PIO_INTR_STATUS_ENABLE) (PIO + 0x24) .....	235
7.5.11 PIO Interrupt Signal Enable (PIO_INTR_SIGNAL_ENABLE) (PIO + 0x28) .....	237
7.5.12 PIO Interrupt Force (PIO_INTR_FORCE) (PIO + 0x2C).....	239
7.5.13 PIO Control (PIO_CONTROL) (PIO + 0x30) .....	240
7.6 Ring Headers Specific Registers .....	241
7.6.1 Ring Headers Preamble (RHS_CONTROL) (RHS + 0x0).....	242
7.6.2 Ring Header 0 Offset (RH0_OFFSET) (RHS + 0x4) .....	243
7.6.3 Ring Header 1 Offset (RH1_OFFSET) (RHS + 0x8) .....	243
7.6.4 Ring Header 2 Offset (RH2_OFFSET) (RHS + 0xC) .....	243
7.6.5 Ring Header 3 Offset (RH3_OFFSET) (RHS + 0x10) .....	243
7.6.6 Ring Header 4 Offset (RH4_OFFSET) (RHS + 0x14) .....	244

7.6.7	Ring Header 5 Offset (RH5_OFFSET) (RHS + 0x18) .....	.244
7.6.8	Ring Header 6 Offset (RH6_OFFSET) (RHS + 0x1C) .....	.244
7.6.9	Ring Header 7 Offset (RH7_OFFSET) (RHS + 0x20) .....	.244
7.6.10	Ring Header Descriptor (RH+) .....	.245
7.7	Extended Capabilities Registers .....	268
7.7.1	Extended Capability Header (EXTCAP_HEADER) .....	.269
7.7.2	Hardware Identification (ID = 0x01) .....	.270
7.7.3	Controller Config (ID = 0x02) .....	.271
7.7.4	Multi-Bus Instance (ID = 0x03) .....	.272
7.7.5	Target IBI Credit Counters (ID = 0x0A) .....	.274
7.7.6	Dead Bus Recovery (ID = 0x0B) .....	.280
7.7.7	Debug Specific (ID = 0x0C) .....	.283
7.7.8	Scheduled Commands for Handler Type 1 (ID = 0x0D) .....	.292
7.7.9	Scheduled Commands for Handler Type 2 (ID = 0x0E) .....	.305
7.7.10	Scheduled Commands for Handler Type 3 (ID = 0x0F) .....	.320
7.7.11	Standby Controller Mode (ID = 0x12) .....	.336
7.7.12	Target-Specific (ID = 0xB0...0xBF) .....	.360
7.7.13	Vendor-Specific Extended Capabilities (ID = 0xC0...0xCF) .....	.360
<b>8</b>	<b>Data Structures.....</b>	<b>361</b>
8.1	Device Address Table (DAT) .....	361
8.1.1	DAT Usage for Transfer Commands .....	.365
8.1.2	DAT Addressing and Parity Bit .....	.365
8.2	Device Characteristic Table (DCT) .....	366
8.3	Transfer Descriptor .....	367
8.3.1	Data Buffer Descriptor .....	.369
8.4	Command Descriptor .....	371
8.4.1	Address Assignment Command .....	.373
8.4.2	Internal Control Command .....	.380
8.4.3	Common Aspects of Transfer Commands .....	.396
8.5	Response Descriptor .....	397
8.6	IBI Status Descriptor .....	399
8.6.1	Common Usage .....	.402
8.6.2	Usage for Regular IBIs .....	.402
8.6.3	Usage for Hot-Join Requests and Controller-Role Requests .....	.403
8.6.4	Usage for Target IBI Credit Counter Updates .....	.403
8.6.5	Usage for Scheduled Command Execution Reports .....	.404
8.6.6	Usage for Auto-Command Read Data .....	.405
8.6.7	Usage for Broadcast CCCs Received in Standby Controller Mode .....	.406
8.7	Memory Descriptor .....	407
<b>9</b>	<b>System Bus Implementations .....</b>	<b>409</b>
9.1	System Bus Requirements .....	409
9.2	Implementation Specifics .....	410
9.2.1	PCI and PCI-Compatible Buses .....	.410
9.2.2	Universal Serial Bus (USB) .....	.412
9.2.3	AMBA Interfaces .....	.414
<b>Annex A</b>	<b>Implementation Guidance (Informative) .....</b>	<b>415</b>
A.1	Relationship to the I3C Transfer Command/Response Interface (TCRI) Specification .....	415
A.2	Variable Data Rates and Queue/FIFO Fill Levels .....	415
<b>Participants .....</b>		<b>417</b>

## Figures

Figure 1 Color Coding Scheme.....	9
Figure 2 I3C System Overview.....	11
Figure 3 Example Host Controller Implementation .....	20
Figure 4 Example of Rings Used for Host-Initiated Transfers .....	27
Figure 5 Rings Used for IBI.....	28
Figure 6 Scatter-Gather for Ring Context Pointers .....	39
Figure 7 Example Flow for Initial Bus Enumeration .....	42
Figure 8 Address Assign Flow .....	43
Figure 9 Device State FSM.....	45
Figure 10 Controller-Role Request (CRR) / In-Band Interrupt (IBI) Autodisable Flow .....	47
Figure 11 PIO Section FSM .....	53
Figure 12 Command Queue FSM .....	57
Figure 13 Response Queue FSM.....	59
Figure 14 IBI Queue Operation for Large and Small Data IBIs.....	61
Figure 15 IBI Queue FSM .....	62
Figure 16 Command Ring FSM .....	72
Figure 17 Response Ring FSM .....	74
Figure 18 IBI Ring Data Chunk Concept.....	76
Figure 19 IBI Ring FSM .....	77
Figure 20 Ring Header Registers .....	82
Figure 21 Transfer in DMA Mode .....	93
Figure 22 Data Payload Byte Ordering (Little Endian).....	95
Figure 23 Data Payload Byte Ordering (Big Endian).....	95
Figure 24 IBI Processing in DMA Mode .....	103
Figure 25 Controller Timestamp Counters for IBI Event.....	104
Figure 26 Target Credit IBI Update FSM.....	108
Figure 27 Auto-Command Generation Flow.....	111
Figure 28 Logical Dependencies Among Interrupt-Related Registers .....	123
Figure 29 Example Host Controller Implementation with Scheduled Commands Logic.....	132
Figure 30 Register Layout for Scheduled Commands Extended Capability Structure (Handler Type 1).....	139
Figure 31 Register Layout for Scheduled Commands Extended Capability Structure (Handler Type 2).....	141
Figure 32 Register Layout for Scheduled Commands Extended Capability Structure (Handler Type 3).....	143
Figure 33 Example Host Controller Implementation with Secondary Controller Logic .....	146
Figure 34 Controller Role FSM with Minimal Secondary Controller Capabilities.....	148
Figure 35 Controller Role FSM with Support for Optional Initialization as Secondary Controller .....	149
Figure 36 Flow to Transition to Standby Controller Mode .....	164
Figure 37 Secondary Controller Broadcast CCC Data Format for IBI Event .....	171
Figure 38 Flow for Transition to Active Controller Mode, Part 1 .....	177
Figure 39 Flow for Transition to Active Controller Mode, Part 2 .....	178
Figure 40 Flow for Rejecting GETACCCR in Standby Controller Mode.....	179

Figure 41 Example I3C System with Dead Bus Recovery.....	181
Figure 42 Example Host Controller Implementation with Dead Bus Recovery Mechanism .....	182
Figure 43 I3C Host Controller Interface Overview.....	185
Figure 44 Concept of Linked List of Extended Capability Structures.....	188
Figure 45 Target IBI Credit Counters Structure Example .....	274
Figure 46 Scheduled Commands Structure Example (Handler Type 1).....	293
Figure 47 Scheduled Commands Structure Example (Handler Type 2).....	306
Figure 48 Scheduled Commands Structure Example (Handler Type 3).....	321
Figure 49 Use of Transfer Descriptor.....	368
Figure 50 Overview of Supported Command Types for Command Descriptor (Part 1) .....	372
Figure 51 Overview of Supported Command Types for Command Descriptor (Part 2) .....	372

## Tables

Table 1 Host Controller Parameters .....	22
Table 2 Queues and Rings Available in PIO and DMA Modes .....	25
Table 3 Scatter-Gather Support for Rings and Other Contexts .....	40
Table 4 Pointers for Command / Response Ring Pair .....	70
Table 5 Pointers for IBI Ring Pair.....	75
Table 6 Actions and States for ABORT Bit in Register HC_CONTROL.....	98
Table 7 Interrupt Register Sets.....	123
Table 8 Supported Scheduled Command Capability Handler Types.....	133
Table 9 Defined Interrupt Events Reported for Controller Role Handoff Procedure .....	161
Table 10 Possible Interrupt Events Reported for Accepting or Not Accepting Controller Role.....	176
Table 11 Register Memory Access Conventions .....	183
Table 12 Register Map: Capabilities and Operation Registers.....	187
Table 13 Extended Capability Mandatory Registers .....	189
Table 14 Ring Header Preamble .....	191
Table 15 Ring Header (Per Ring Bundle) .....	191
Table 16 Register Map: PIO Access Area .....	192
Table 17 HCI Version (HCI_VERSION) Register .....	193
Table 18 Host Controller Control (HC_CONTROL) Register .....	194
Table 19 Controller Device Address (CONTROLLER_DEVICE_ADDR) Register.....	199
Table 20 Host Controller Capabilities (HC_CAPABILITIES) Register.....	201
Table 21 Reset Control (RESET_CONTROL) Register .....	204
Table 22 Present State (PRESENT_STATE) Register.....	206
Table 23 Interrupt Status (INTR_STATUS) Register.....	207
Table 24 Interrupt Status Enable (INTR_STATUS_ENABLE) Register .....	208
Table 25 Interrupt Signal Enable (INTR_SIGNAL_ENABLE) Register.....	209
Table 26 Interrupt Force (INTR_FORCE) Register .....	210
Table 27 Device Address Table Section Offset (DAT_SECTION_OFFSET) Register .....	211
Table 28 Device Characteristics Table Section Offset (DCT_SECTION_OFFSET) Register .....	212
Table 29 Ring Headers Section Offset (RING_HEADERS_SECTION_OFFSET) Register.....	213
Table 30 PIO Section Offset (PIO_SECTION_OFFSET) Register .....	213
Table 31 Extended Capabilities Section Offset (EXT_CAPS_SECTION_OFFSET) Register .....	214
Table 32 Internal Control Command Subtype Support (INT_CTRL_CMDS_EN) Register .....	215
Table 33 IBI Notify Control (IBI_NOTIFY_CTRL) Register .....	216
Table 34 IBI Data Abort Control (IBI_DATA_ABORT_CTRL) Register .....	218
Table 35 Device Context Address Low (DEV_CTX_BASE_LO) Register.....	220
Table 36 Device Context Base Address High (DEV_CTX_BASE_HI) Register.....	221
Table 37 Device Context Scatter-Gather Support (DEV_CTX_SG) Register.....	221
Table 38 Rx Data Port (RX_DATA_PORT) Register.....	223
Table 39 Tx Data Port (TX_DATA_PORT) Register .....	223
Table 40 IBI Port (IBI_PORT) Register.....	224

Table 41 Queue Threshold Control (QUEUE_THLD_CTRL) Register .....	.225
Table 42 Data Buffer Threshold Control (DATA_BUFFER_THLD_CTRL) Register .....	.227
Table 43 Queue Size (QUEUE_SIZE) Register .....	.229
Table 44 Alternate Queue Size (ALT_QUEUE_SIZE) Register .....	.231
Table 45 PIO Interrupt Status (PIO_INTR_STATUS) Register .....	.233
Table 46 PIO Interrupt Status Enable (PIO_INTR_STATUS_ENABLE) Register .....	.235
Table 47 PIO Interrupt Signal Enable (PIO_INTR_SIGNAL_ENABLE) Register .....	.237
Table 48 PIO Interrupt Force (PIO_INTR_FORCE) Register .....	.239
Table 49 PIO Control (PIO_CONTROL) Register .....	.240
Table 50 Ring Headers Preamble (RHS_CONTROL) Register .....	.242
Table 51 Ring Header Descriptor Offset (RH0_OFFSET) Register .....	.243
Table 52 Ring Header Descriptor Offset (RH1_OFFSET) Register .....	.243
Table 53 Ring Header Descriptor Offset (RH2_OFFSET) Register .....	.243
Table 54 Ring Header Descriptor Offset (RH3_OFFSET) Register .....	.243
Table 55 Ring Header Descriptor Offset (RH4_OFFSET) Register .....	.244
Table 56 Ring Header Descriptor Offset (RH5_OFFSET) Register .....	.244
Table 57 Ring Header Descriptor Offset (RH6_OFFSET) Register .....	.244
Table 58 Ring Header Descriptor Offset (RH7_OFFSET) Register .....	.244
Table 59 Command/Response Ring Control (CR_SETUP) Register .....	.246
Table 60 IBI Ring Control (IBI_SETUP) Register .....	.249
Table 61 Chunk Control (CHUNK_CONTROL) Register .....	.250
Table 62 Ring Interrupt Status (RH_INTR_STATUS) Register .....	.251
Table 63 Ring Interrupt Status Enable (RH_INTR_STATUS_ENABLE) Register .....	.253
Table 64 Ring Interrupt Signal Enable (RH_INTR_SIGNAL_ENABLE) Register .....	.255
Table 65 Ring Interrupt Force (RH_INTR_FORCE) Register .....	.256
Table 66 Ring Control Status (RH_STATUS) Register .....	.257
Table 67 Ring Control (RH_CONTROL) Register .....	.258
Table 68 Ring Operation 1 (RH_OPERATION1) Register .....	.259
Table 69 Ring Operation 2 (RH_OPERATION2) Register .....	.259
Table 70 Command Ring Base Address Low (RH_CMD_RING_BASE_LO) Register .....	.260
Table 71 Command Ring Base Address High (RH_CMD_RING_BASE_HI) Register .....	.260
Table 72 Response Ring Base Address Low (RH_RESP_RING_BASE_LO) Register .....	.261
Table 73 Response Ring Base Address High (RH_RESP_RING_BASE_HI) Register .....	.261
Table 74 IBI Status Ring Base Address Low (RH_IBI_STATUS_RING_BASE_LO) Register .....	.262
Table 75 IBI Status Ring Base Address High (RH_IBI_STATUS_RING_BASE_HI) Register .....	.262
Table 76 IBI Data Ring Base Address Low (RH_IBI_DATA_RING_BASE_LO) Register .....	.263
Table 77 IBI Data Ring Base Address High (RH_IBI_DATA_RING_BASE_HI) Register .....	.263
Table 78 Command Ring Scatter-Gather Support (RH_CMD_RING_SG) Register .....	.264
Table 79 Response Ring Scatter-Gather Support (RH_RESP_RING_SG) Register .....	.265
Table 80 IBI Status Ring Scatter-Gather (RH_IBI_STATUS_RING_SG) Register .....	.266
Table 81 IBI Data Ring Scatter-Gather Support (RH_IBI_DATA_RING_SG) Register .....	.267
Table 82 Extended Capability IDs .....	.268

Table 83 Extended Capability Header (EXTCAP_HEADER) Register.....	269
Table 84 Component Manufacturer (COMP_MANUFACTURER) Register .....	270
Table 85 Component Version (COMP_VERSION) Register .....	270
Table 86 Component Type (COMP_TYPE) Register .....	270
Table 87 Controller Config (CONTROLLER_CONFIG) Register.....	271
Table 88 Bus Controller Instance Count (BUS_CTRL_INSTANCE_COUNT) Register .....	272
Table 89 Bus Controller Instance ## Offset (BUS_CTRL_INSTANCE_##_OFFSET) Register .....	273
Table 90 Target Credit Table (TARGET_CREDIT_TABLE) Register .....	275
Table 91 Target Credit Configuration (TARGET_CREDIT_CONFIG) Register.....	277
Table 92 Target Credit Count ## (TARGET_CREDIT_##) Register .....	278
Table 93 Dead Bus Recovery Engage (DBR_ENGAGE) Register .....	280
Table 94 Queue Status Level (QUEUE_STATUS_LEVEL) Register.....	283
Table 95 Data Buffer Status Level (DATA_BUFFER_STATUS_LEVEL) Register.....	284
Table 96 Present State Debug (PRESENT_STATE_DEBUG) Register .....	285
Table 97 MX Error Counters (MX_ERROR_COUNTERS) Register .....	289
Table 98 Scheduled Commands Debug (SCHED_CMDS_DEBUG) Register .....	290
Table 99 Schedule Capabilities (SCHEDULE_CAPABILITIES) Register.....	295
Table 100 Schedule Table Layout (SCHEDULE_TABLE_LAYOUT) Register.....	296
Table 101 Schedule Configuration (SCHEDULE_CONFIG) Register .....	298
Table 102 Schedule Table Entry ## Mask (SCHEDULE_TABLE_##_MASK) Register .....	300
Table 103 Schedule Table Entry ## Config (SCHEDULE_TABLE_##_CONFIG) Register .....	301
Table 104 Schedule Capabilities (SCHEDULE_CAPABILITIES) Register.....	308
Table 105 Schedule Layout (SCHEDULE_LAYOUT) Register.....	309
Table 106 Schedule Configuration (SCHEDULE_CONFIG) Register .....	311
Table 107 Schedule Sequence Mask (SCHEDULE_SEQ_MASK) Register .....	312
Table 108 Schedule Sequence Config (SCHEDULE_SEQ_CONFIG) Register .....	314
Table 109 Schedule Sequence Targets ## (SCHEDULE_SEQ_TARGETS_##) Register .....	317
Table 110 Schedule Capabilities (SCHEDULE_CAPABILITIES) Register.....	322
Table 111 Schedule Layout (SCHEDULE_LAYOUT) Register.....	323
Table 112 Schedule Configuration (SCHEDULE_CONFIG) Register .....	325
Table 113 Schedule Buffer Mask (SCHEDULE_BUF_MASK) Register.....	326
Table 114 Schedule Buffer Config (SCHEDULE_BUF_CONFIG) Register .....	327
Table 115 Schedule Buffer Descriptor (SCHEDULE_BUF_DESCR) Register.....	330
Table 116 Schedule Buffer Base Address Low (SCHEDULE_BUF_BASE_LO) Register .....	332
Table 117 Schedule Buffer Base Address High (SCHEDULE_BUF_BASE_HI) Register .....	333
Table 118 Register Map of Extended Capability Structure for Standby Controller Mode .....	336
Table 119 Standby Controller Control (STBY_CR_CONTROL) Register.....	337
Table 120 Standby Controller Device Address (STBY_CR_DEVICE_ADDR) Register.....	343
Table 121 Standby Controller Capabilities (STBY_CR_CAPABILITIES) Register .....	345
Table 122 Standby Controller Status (STBY_CR_STATUS) Register .....	347
Table 123 Standby Controller Device Characteristics (STBY_CR_DEVICE_CHAR) Register .....	349
Table 124 Standby Controller Device Characteristics (STBY_CR_DEVICE_PID_LO) Register.....	350

Table 125 Standby Controller Interrupt Status (STBY_CR_INTR_STATUS) Register .....	352
Table 126 Standby Controller Interrupt Signal Enable (STBY_CR_INTR_SIGNAL_ENABLE) Register .....	355
Table 127 Interrupt Force (STBY_CR_INTR_FORCE) Register.....	357
Table 128 CCC Auto-Response Config Get Capabilities (STBY_CR_CCC_CONFIG_GETCAPS).....	358
Table 129 CCC Auto-Response Config Target Reset Action (STBY_CR_CCC_CONFIG_RSTACT_PARAMS) .....	359
Table 130 Device Address Table Structure .....	362
Table 131 Device Characteristic Table Structure .....	366
Table 132 Data Buffer Descriptor Structure.....	370
Table 133 Supported Command Types for Command Descriptor.....	371
Table 134 Address Assignment Command Structure .....	373
Table 135 Internal Control Command Structure .....	381
Table 136 Internal Control Command Fields for Ring Bundle Lock/Unlock.....	382
Table 137 Internal Control Command Structure: Fields for 0x7E Sub Command .....	383
Table 138 Internal Control Command Fields for Device Context Update .....	384
Table 139 Internal Control Command for Target Reset Pattern .....	385
Table 140 Internal Control Command for Controller SDA Recovery or Bus Reset Procedure .....	386
Table 141 Internal Control Command for Enable End Transfer Termination and HDR Mode Configuration .....	389
Table 142 Internal Control Command for Controller Role Handoff Procedure with GETACCCR CCC.....	391
Table 143 Response Descriptor Status for Controller Role Handoff Procedure with GETACCCR CCC .....	392
Table 144 Internal Control Command for Attempt Dead Bus Recovery.....	394
Table 145 Response Descriptor Status for Controller Role Handoff Procedure with GETACCCR CCC .....	395
Table 146 Summary of Response Descriptor Structure Fields .....	397
Table 147 IBI Status Descriptor Structure .....	399
Table 148 Memory Descriptor Structure.....	407

## Release History

Date	Version	Description
04-Apr-2018	v1.0	Initial Board adopted release.
20-May-2021	v1.1	Board adopted release.
12-Apr-2023	v1.2	Board adopted release.

This page intentionally left blank

## 1 Introduction

The proliferation of sensors in mobile wireless and mobile-influenced products has created significant design challenges. The number of sensors in platforms increases each year, and sensors are becoming crucial for enabling new use cases and for platform power management.

The standardization of interfaces and common software support allow for more reliable hardware and software. In platform design, the use of common components allows designers to focus efforts on actual sensor applications, rather than the interfaces.

If no consistent method for interfacing to I3C were available, then every platform vendor would be faced with designing and enabling their own interface to support I3C. In addition to the main interface other signals may be needed, such as dedicated interrupts, chip select signals, and enable and sleep signals. This increases the required number of Host GPIOs, and that in turn drives up system cost with more Host package pins and more PCB layers. As time passes and the number of sensors increases, this becomes increasingly difficult to support and manage.

The MIPI I3C interface has been 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. I3C is compatible with existing Legacy I<sup>2</sup>C Devices, with feature limitations. I3C defines the idea of Controller Devices and Target Devices (i.e., I3C Devices with the role of Controller and/or Target). I3C also allows for multiple Controller-capable Devices (i.e., one Primary Controller and optionally one or more Secondary Controllers) in the Bus topology.

This I3C HCI Specification describes the system interface exposed by hardware implementing I3C Primary Controller Devices, i.e., the Host Controller Interface for the MIPI I3C protocol [[MIPI02](#)].

### 1.1 Scope

This Specification contains sufficient detail to develop both hardware and software for an I3C Host Controller. It includes an Architectural Overview, the Theory of Operation, and includes definitions of the Register Set and Data Structures that define the I3C Host Controller.

The reader is assumed to be familiar with the I3C Specification [[MIPI02](#)] And the I3C TCRI Specification [[MIPI06](#)]. Concepts described in these other specifications will not be repeated here, except where necessary for proper understanding.

Although this Specification is OS-agnostic, it does reflect implementation considerations relevant for the Linux and Windows operating systems.

### 1.2 Purpose

The purpose of this Specification is to reduce the development and maintainability cost of I3C by creating a standard definition for the Host Controller Interface. This allows a single OS Driver (aka. ‘in-box Driver’) to support I3C Hardware from several vendors. This Specification allows for vendor-specific extensions or other improvements, which are treated as additional capabilities claimed by the Host Controller.

The target audience of the document are developers of Primary Controller (Host Controller) hardware, and developers of I3C Host Controller software.

## 2 Terminology

### 2.1 Use of Special Terms

The MIPI Alliance has adopted Section 13.1 of the *IEEE Standards Style Manual*, which dictates use of the words “shall”, “should”, “may”, and “can” in the development of documentation, as follows:

The word *shall* is used to indicate mandatory requirements strictly to be followed in order to conform to the Specification and from which no deviation is permitted (*shall* equals *is required to*).

The use of the word *must* is deprecated and shall not be used when stating mandatory requirements; *must* is used only to describe unavoidable situations.

The use of the word *will* is deprecated and shall not be used when stating mandatory requirements; *will* is only used in statements of fact.

The word *should* is used to indicate that among several possibilities one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain course of action is deprecated but not prohibited (*should* equals *is recommended that*).

The word *may* is used to indicate a course of action permissible within the limits of the Specification (*may* equals *is permitted to*).

The word *can* is used for statements of possibility and capability, whether material, physical, or causal (*can* equals *is able to*).

All sections are normative, unless they are explicitly indicated to be informative.

### 2.2 Definitions

**a priori:** Knowledge resulting from theoretical deduction, not from observation or experience (Latin).

**ACK:** Short for “acknowledge”. See also NACK.

**Active Controller:** The I3C Device that presently has control of the I3C Bus (i.e., that has the Controller Role). Also refers to the operating mode of a Host Controller that has the Controller Role, and can initiate transfers to I3C Targets or I3C Secondary Controllers.

**Address Arbitration:** Process for determining arbitrated Addresses to resolve contention.

**Address:** A set of bits designating a Device or the location of a register.

**Arbitrable:** Subject to decision by Arbitration.

**Arbitration:** If two Devices start transmission at the same time, then Arbitration is required to determine Bus control. Arbitration could also be required during a Target transmission if a Controller addresses multiple Target Devices. Arbitration is required when a Controller sends an address and when a Target sends an In-Band Interrupt.

**Atomic:** (Relating to an operation or transaction) A sequence comprising a set of smaller operations, provided in a particular order, which must either be executed fully from start to finish without interruption; executed in part (if applicable to the sequence), stopping only at designated safe points of interruption, and not allowed to continue after any such interruption, due to unintended consequences that would induce behavior different from that which would be induced by continuous execution; or cancelled entirely after any such interruption. In cases of an interruption, any subsequent smaller operations that might not have been executed after the interruption (e.g., when some smaller operations were not yet received before the determination of the end of the sequence was known) must either be nullified or not executed.

**Broadcast:** Refers to an Address or command that is transmitted to multiple Target Devices.

74     **Bundle:** See Ring Bundle.

75     **Bus Controller Logic:** An internal component comprising dedicated logic and internal transfer  
76     mechanisms, integrated within the Host Controller, that acts as the Controller of the I3C Bus in Active  
77     Controller mode. Typically contains internal FIFOs, clock logic, status, and other interfaces under the  
78     control of the Host Controller to process Transfer Commands from the active Command Queue/Ring,  
79     handle Interrupt Requests and perform other necessary tasks in the role of Active Controller. If the Host  
80     Controller also supports Standby Controller mode, then the Bus Controller Logic is inactive.

81     **Characteristics:** Quantification of a Device's available features and capabilities.

82     **Combo:** A two phased transfer operation, typically consisting of the combination of two transfers as a  
83     single unit. Usually structured as either a Write + Write operation, or a Write + Read operation. The  
84     transfers for these phases might be separated by specific framing specific to the I3C Mode(s) used for each  
85     phase. Typically, the first phase is a shorter transfer which informs a Device to prepare for the second  
86     phase.

87     **Command Descriptor:** Structure used to define I3C Command, CCC or Transfer.

88     **Common Command Codes (CCC):** Globally supported commands to be transmitted either directly to a  
89     specific I3C Target Device or to all I3C Target Devices simultaneously.

90     **Controller:** A reference to the I3C Bus Device that is controlling the Bus.

91     **Controller Role:** Control of the I3C Bus, in an Active Controller role. See Active Controller.

92     **Controller Role Request:** A method whereby a Controller-capable Device (i.e., a Secondary Controller)  
93     requests to become the next Active Controller of the Bus, using a form of interrupt request (i.e., In-Band  
94     Interrupt).

95     **Critical Section:** A protected phase of operation where the Host Controller's Command/Response interface  
96     is only accessed by one software context (i.e., a Driver thread or other similar construct) and is expected to  
97     be blocked from concurrent access by other software contexts. Typically, this software context is dedicated  
98     to performing a specific task for the defined purpose, and expects the Command/Response interface to  
99     operate with a similar modality. In most cases, the time window for a critical section is finite in duration  
100    and limited in scope.

101    **Data Port:** Register that allows for placing or/and receiving transfer/CCC payload data, in PIO Mode.

102    **Deep Sleep:** A lower-power state in which the Bus Controller Logic retains its configuration but does not  
103    typically drive transactions on the Bus. In Standby Controller mode, this also refers to the state in which the  
104    Secondary Controller Logic retains its configuration and Dynamic Address but does not respond to Bus  
105    activity, and typically does not attempt to notify the Host of any actions taken by the Active Controller of  
106    the Bus. In such a state, the Host might also be in a lower-power state, and might also be unable to respond  
107    to Bus activity (if it were monitored).

108    **Device:** A Controller or Target.

109    **Device Address Table:** Table that contains information on all Target Devices addressable on the I3C Bus.

110    **Device Characteristics Table:** Transient table used to automatically retrieve Target Device information on  
111    Target Device enumeration.

112    **Device ID:** Defines a Device's characteristic or function within a sensor system.

113    **DMA Mode:** An operating mode that relies on a system bus that provides direct memory access (DMA)  
114    capability and internal DMA engines to enable autonomous Host-Controller-initiated memory access  
115    requests, in order to drive transactions without a direct software involvement.

116    **Doorbell:** Method of notifying the Host Controller when it must take an action. In this Specification,  
117    Doorbells are implemented as register write operations.

118    **Driver:** Software entity (Driver) controlling Host Controller hardware.

119    **DWORD:** A 32-bit data word.

120   **Dynamic Address:** A Device Address that is assigned or allocated during initialization of the I3C Bus, or  
121    subsequently as needed. Usually occurs after power up.

122   **Extended Capability:** An optional section of registers (i.e., Register Set) of a Host Controller providing  
123    additional capabilities, optional modes, or features. Extended Capability structures are discoverable, have  
124    defined register layouts, and are arranged as a linked list of structures.

125   **Frame:** A Frame begins with a START, followed by the Address of the targeted Target(s), Data, and finally  
126    a STOP.

127   **HCI:** Host Controller Interface

128   **HDR-DDR:** HDR Double Data Rate

129   **HDR-BT:** HDR Bulk Transfer

130   **High Data Rate (HDR):** High Data Rate modes that achieve higher speed by transferring data on both  
131    clock edges.

132   **High:** A signal level of logical “1”.

133   **Host:** See Primary Controller.

134   **Host Controller:** Hardware entity that implements this I3C HCI specification and provides I3C Controller  
135    (and optional I3C Secondary Controller) functionality.

136   **Hot-Join:** Target Devices that join the Bus after it is already started, whether because they were not  
137    powered previously or because they were physically inserted into the Bus; the Hot-Join mechanism allows  
138    the Target to notify the Controller that it is ready to get a Dynamic Address.

139   **IBI Status Descriptor:** Structure used to define incoming IBI.

140   **In-Band Interrupt:** Interrupt from Target Device on I3C Bus without a separate pin connection.

141   **I<sup>2</sup>C Device:** A Controller or Target that meets the requirements of the I<sup>2</sup>C Specification [*NXP01*].

142   **I3C Bus:** The physical and logical implementation of the SCL and SDA lines according to I3C  
143    Specification [*MIP102*].

144   **I3C Device:** A Controller or Target that meets the requirements of the I3C Specification [*MIP102*].

145   **I3C Bus Controller Logic:** See Controller Logic.

146   **I3C Secondary Controller Logic:** See Secondary Controller Logic.

147   **I3C Target:** See Target.

148   **I3C Target Transaction Interface:** An optional capability, outside of the scope of this I3C HCI  
149    Specification, that allows the Secondary Controller Logic to participate in more types of I3C transactions  
150    (i.e., above those that are already defined in this I3C HCI Specification). For example, SDR Private Write  
151    and Private Read transfers and additional CCC flows.

152   **Note:**

153    *Although this I3C HCI Specification does not define or attempt to describe additional details of an*  
154    *I3C Target Transaction Interface, implementers may choose to add an I3C Target Transaction*  
155    *Interface as an optional extended capability. This interface would work along with the Secondary*  
156    *Controller Logic, as an adjunct to its capabilities.*

157   **Legacy I<sup>2</sup>C:** I3C maintains the industry standard architecture of I<sup>2</sup>C and supports existing I<sup>2</sup>C Target  
158    Devices. I3C does not support I<sup>2</sup>C Bus Controllers.

159   **Low:** A signal level of logical “0”.

160   **Memory Access Engine:** An external DMA engine that processes individual write and read requests to  
161    regions of Host system memory via its connected bus; or a similar internal component that places access  
162    requests to the Host system bus’s existing memory access unit on behalf of the Memory Access Interface.

163     **Memory Access Interface:** An internal component that drives autonomously-initiated transfers via  
164     Memory Access Engines for specific uses of Host-allocated memory regions. For DMA Mode, this allows  
165     the Ring Controller to access the configured memory regions for Rings, as well as the enqueued data  
166     structures for individual I3C Transfer Commands, that describe where the transfer data will be written or  
167     read.

168     **Message:** A packetized communication between Devices.

169     **Minimal Standby Controller:** A Host Controller implementation that supports only the minimum required  
170     features and capabilities of an I3C Secondary Controller, and can operate in Standby Controller mode.

171     **MIPI Manufacturer ID:** A two byte (16 bit) unique identifier for a vendor of a MIPI compliant Device  
172     [[MIPI01](#)].

173     **Mixed Fast Bus:** I3C Bus topology with both I<sup>2</sup>C and I3C Devices present on the I3C Bus, where the I<sup>2</sup>C  
174     Devices have a true I<sup>2</sup>C 50 ns Spike Filter on the SCL line.

175     **Mixed Slow and Limited Bus:** I3C Bus topology with both I<sup>2</sup>C and I3C Devices present on the I3C Bus,  
176     where the I<sup>2</sup>C Devices do not have a true I<sup>2</sup>C 50 ns Spike Filter on the SCL line.

177     **Mode:** (Does not apply to operating modes, such as DMA Mode and PIO Mode.) I3C defined data transfer  
178     methods, namely: Legacy I<sup>2</sup>C Mode; Single Data Rate Mode (SDR); and various High Data Rate (HDR)  
179     Modes including Dual Data Rate Mode (HDR-DDR), Bulk Transfer Mode (HDR-BT), Ternary Symbol  
180     Legacy Mode (HDR-TSL), and Ternary Symbol for Pure Bus Mode (HDR-TSP).

181     **Multi-Drop:** A Bus that communicates through a process of Arbitration to determine which Device sends  
182     information at any point. The other Devices listen for data they are intended to receive.

183     **NACK:** Short for “not acknowledge”, which means No ACK was asserted. See also ACK.

184     **Offline Capable:** An Offline Capable Device is able to disconnect from the physical I3C Bus and/or is able  
185     to ignore I3C traffic on the I3C Bus. A Device’s Offline capability is one of the capabilities reflected in its  
186     Bus Characterization Register.

187     **Open-Drain:** High-Z with an active Pull-Down. Typically used in conjunction with a passive Pull-Up.

188     **Optional-Normative:** Describes a feature or capability that is optional for the implementer to support; but  
189     if the implementer chooses to support it, then the requirements for this feature/capability are normatively  
190     defined in a specific manner by this I3C HCI Specification.

191     **PIO Mode:** An operating mode that relies on Host-driven programming of the Host Controller’s registers  
192     and internal buffers/queues to enqueue all transactions, which requires a high degree of direct software  
193     involvement.

194     **Primary Controller:** Controller that has overall control of the I3C Bus, including control and handoff to  
195     Secondary Controllers.

196     **Pure Bus:** A Bus topology with only I3C Devices present. No I<sup>2</sup>C Devices are permitted on a Pure Bus.

197     **Push-Pull:** Active Pull-Down and active Pull-Up on output driver.

198     **QWORD:** Quad-Word (8 Bytes)

199     **Read-Type Transfer:** An operation, typically having a single phase, in which a Device on the I3C Bus is  
200     addressed and then required to either acknowledge or refuse a read transfer, based on transfer parameters  
201     sent on the I3C Bus. If the Device acknowledges the transfer, then it shall immediately drive data bytes on  
202     the data line (i.e., SDA) which are received by the Host Controller and made available for the Host to read.

203     **Register Map:** A number of registers mapped to software use.

204     **Register Set:** A number of registers exposed by Host Controller.

205     **Repeated START:** Two or more instances of a START in a row without an intervening STOP. A Repeated  
206     START is used in circumstances where the Controller wishes to continue communicating on the I3C Bus

207 without having to first generate a STOP. In this Specification, a Repeated START is abbreviated as “Sr”.  
208 This is equivalent to Repeated START in I<sup>2</sup>C [NXP01].

209 **Response Descriptor:** Structure used to report Command or Transfer status

210 **Ring:** The Ring is software-allocated memory used for communication between the Host Controller and the  
211 Host Controller Driver. The Ring is a cyclic buffer of structures that are managed and used by transaction  
212 management and maintenance logic (i.e., the Ring Controller) and exposed to the Host via a set of registers  
213 (i.e., the Ring Header).

214 **Ring Bundle:** Concept of Combined Command, Response and IBI Rings. The Target Devices can be  
215 grouped by assigning Ring ID.

216 **Ring Controller:** Transaction management and maintenance logic that enables a Host to enqueue  
217 transactions into the Ring (i.e., the cyclic buffer of structures) and uses the Memory Access Interface to  
218 manage all memory access to/from the Host, in order to serialize and drive transactions to the I3C Bus and  
219 report response status. Arbitrates the servicing of available Command/Response Ring Pairs that have  
220 enqueued transfers waiting to be processed, when multiple Ring Bundles are enabled.

221 **Ring Header:** Register Set exposed by Host Controller to implement single Ring Bundle and implemented  
222 within the Ring Controller.

223 **Ring ID:** Identifier of a Ring Bundle.

224 **Ring Pair:** Combination of two paired Rings. This specification defines Command and Response Ring Pair  
225 or IBI Status and IBI Data Ring Pair.

226 **SDR-Only:** An SDR-Only Device supports only SDR Mode, i.e., does not support any HDR Mode(s).

227 **Scatter-Gather Buffer:** A memory buffer that is described by a Scatter-Gather List, and may be the union  
228 of multiple allocations (i.e., blocks) of memory, each of which is physically contiguous.

229 **Scatter-Gather List:** List of physically contiguous memory blocks that form a memory buffer (i.e., a  
230 Scatter-Gather Buffer). A data payload may be provided (i.e., stored) either in a single physically  
231 contiguous memory block, or in a number of such memory blocks; the term Scatter-Gather List refers to the  
232 latter method.

233 **Secondary Controller:** Controller-capable I3C Device that controls the I3C Bus only after receiving  
234 permission (i.e., after accepting the Controller Role) from the Primary Controller. Control of the Bus might  
235 be temporary; if so, such a Device typically passes the Controller Role back to either the Primary  
236 Controller, or to another Controller-capable Device.

237 **Secondary Controller Logic:** The additional logic and internal transfer mechanisms, as an integrated  
238 feature with a Host Controller that can act as a Secondary Controller on the I3C Bus (i.e., in Standby  
239 Controller mode). This logic is active when the Controller Role has been passed to another I3C Controller-  
240 capable Device, or when joining the I3C Bus as a Secondary Controller (i.e., not the Active or Primary  
241 Controller). May be contained within, or implemented as an adjunct to, the Bus Controller Logic.

242 **Single Data Rate (SDR):** Single Data Rate transfers data on only one edge of the clock.

243 **Stall:** The act of the I3C Controller holding the SCL LOW under specific transitory conditions.

244 **Standby Controller:** A Host Controller that supports Secondary Controller Logic and is not presently the  
245 Active Controller of the Bus. Usually defines an operating mode with limited capabilities available to the  
246 Host, in contrast to the capabilities that are possible while operating in Active Controller mode.

247 **START:** START is the I3C Bus condition of a HIGH to LOW transition on the SDA line while the SCL  
248 line remains HIGH. In this Specification, a START is abbreviated as “S”.

249 **Static Address:** A Device Address that is fixed and cannot be changed.

250 **STOP:** STOP is the I3C Bus condition of a LOW to HIGH transition on the SDA line while the SCL line  
251 remains HIGH. In this Specification, a STOP is abbreviated as “P”.

252 **Synchronization:** Coordination of events to operate a system in unison.

253   **Target:** An I3C Device that can only respond to either Common or individual commands from a Controller.  
254   A Target Device cannot typically generate a clock.

255   **Note:**

256   *In earlier versions of the I3C HCI Specification the term “Slave” was used for such Devices,*  
257   *because that was the term that earlier versions of the I3C Specification used. This term has*  
258   *since been changed to “Target”, and as a result this version of the I3C HCI Specification now*  
259   *also uses the term “Target”. The technical definition of such a Device, and of its role on an*  
260   *I3C Bus, remain unchanged; only the terminology itself has been changed.*

261   **Ternary Mode:** A term used as reference to I3C HDR-TSP (Ternary Symbol for Pure Bus) or HDR-TSL  
262   (Ternary Symbol Legacy) Modes.

263   **Transfer Command:** Structure used to define a Command or Data Transfer to a Target Device on the I3C  
264   Bus. Defined normatively in the I3C Transfer Command/Response Interface (I3C TCRI) Specification (see  
265   [[MIPI06](#)]).

266   **Transfer Descriptor:** Structure used to define a Command or Data Transfer to a Target Device, including a  
267   Command Descriptor and a pointer to a region of Host system memory that contains data. Only used for  
268   DMA Mode.

269   **Version 1.1.1+ of the I3C Specification:** See [[MIPI05](#)].

270   **Word:** Transmission containing 16 payload bits and two parity bits.

271   **Write-Type Transfer:** An operation, typically having a single phase, in which a Device on the I3C Bus is  
272   addressed and then required to either acknowledge or refuse a write transfer, based on transfer parameters  
273   sent on the I3C Bus. If the Device acknowledges the transfer, it shall then receive data bytes on the data  
274   line (i.e., SDA) which are driven by the Host Controller.

## 2.3 Abbreviations

275	e.g.	For example (Latin: <i>exempli gratia</i> )
276	i.e.	That is (Latin: <i>id est</i> )
277	aka.	Also known as

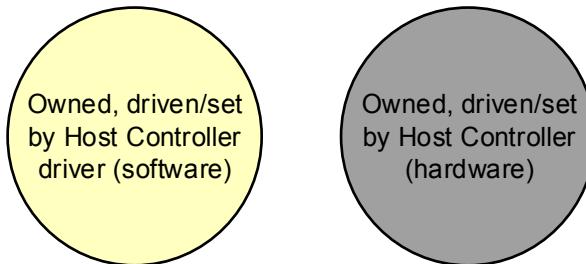
## 2.4 Acronyms

278	ACK	Acknowledge
279	AHB	Advanced High-performance Bus (defined in Arm® AMBA® Specifications [ <a href="#">ARM01</a> ])
280	AMBA	Advanced Microcontroller Bus Architecture [ <a href="#">ARM01</a> ]
281	AXI	Advanced eXtensible Interface (defined in Arm® AMBA® Specifications [ <a href="#">ARM01</a> ])
282	BCR	Bus Characteristics Register
283	CCC	Common Command Code
284	CRC	Cyclic Redundancy Check
285	CRR	Controller-Role Request
286	DAT	Device Address Table
287	DCR	Device Characteristics Register
288	DCT	Device Information and Bus Characteristics Table
289	DDR	Double Data Rate

290	DMA	Direct Memory Access, an I3C HCI transfer Mode (DMA Mode)
291	FSM	Finite State Machine
292	HCI	Host Controller Interface
293	HDR	High Data Rate
294	HDR-BT	HDR Bulk Transfer
295	HDR-DDR	HDR Double Data Rate
296	HDR-TSL	HDR Ternary Symbol Legacy
297	HDR-TSP	HDR Ternary Symbol for Pure Bus (no I <sup>2</sup> C Devices)
298	HJ	Hot-Join
299	I3C TCRI	MIPI I3C Transfer Command/Response Interface Specification [ <i>MIPI06</i> ]
300	IBI	In-Band Interrupt
301	LSb	Least Significant Bit
302	LSB	Least Significant Byte
303	MDB	Mandatory Data Byte
304	MHz	Mega Hertz
305	MID	MIPI Manufacturer Identification [ <i>MIPI01</i> ]
306	MMIO	Memory Mapped I/O
307	MSb	Most Significant Bit
308	MSB	Most Significant Byte
309	MUX	Multiplexer
310	NACK	Not Acknowledge
311	P	STOP
312	PCI*	A System Bus that maintains logical compatibility with PCI [ <i>PCISIG01</i> ]
313	PIO	Programmable I/O, an I3C HCI transfer Mode (PIO Mode)
314	S	START
315	SCL	Serial Clock
316	SDA	Serial Data
317	SDR	Single Data Rate
318	Sr	Repeated START
319	T	Transition Bit
320	USB	Universal Serial Bus [ <i>USB01</i> ] and [ <i>USB02</i> ]

## 2.5 Color Coding

In some Figures in this Specification, color coding is used to indicate ownership. This includes ownership of state in FSMs, and ownership of registers. Items driven or set by software (i.e., the Host Controller Driver) are indicated with yellow, and items driven or set by the Host Controller are indicated with gray.



**Figure 1 Color Coding Scheme**

321  
322  
323

### 3 References

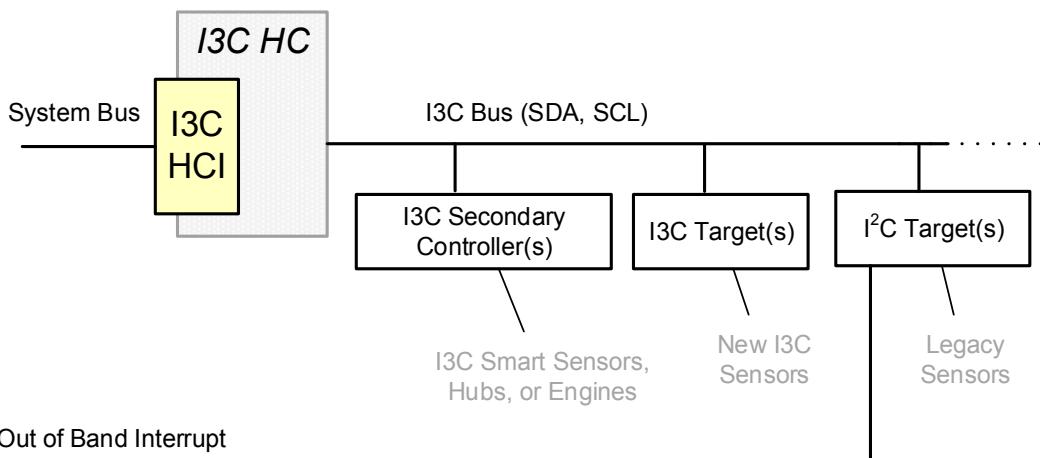
- 325 [ARM01] *Arm® Advanced Microcontroller Bus Architecture (AMBA®)*,  
326 <<https://developer.arm.com/architectures/system-architectures/amba>>, Generation 5,  
327 Arm Limited, 15 October 2019 (and prior).
- 328 [MIPI01] MIPI Alliance, Inc., “MIPI Alliance Manufacturer ID Page”, <<http://mid.mipi.org>>, last  
329 accessed 12 April 2023.
- 330 [MIPI02] *Specification for Improved Inter Integrated Circuit (I3C®)*, Version 1.1.1 including  
331 Errata 01 and Errata 02, MIPI Alliance, Inc., 20 May 2021 (MIPI Board Adopted  
332 20 May 2021, Errata 01 dated 18 February 2022, Errata 02 dated 10 March 2022).
- 333 [MIPI05] *In this Specification, the term “Version 1.1.1+ of the I3C Specification” refers to the most  
334 recently adopted MIPI I3C v1.1.1-based Specifications.*
- 335 **Note:**
- 336 *At the time this I3C HCI v1.2 Specification was adopted, this was the MIPI I3C  
337 v1.1.1 Specification ([MIPI02], available to MIPI Alliance member companies  
338 only) and the MIPI I3C Basic v1.1.1 Specification ([MIPI07], publicly available).*
- 339 [MIPI03] *I3C® Application Note: General Topics (Applies to MIPI I3C v1.1+ and MIPI I3C Basic  
340 v1.1.1+)*, App Note version 1.1, MIPI Alliance, Inc., 27 April 2022  
(MIPI Board approved 27 July 2022).
- 342 [MIPI04] *Discovery and Configuration (DisCo<sup>SM</sup>) Specification for I3C*, Version 1.0,  
343 MIPI Alliance, Inc., 23 January 2019 (MIPI Board Adopted 18 June 2019).
- 344 [MIPI06] *Specification for I3C Transfer Command/Response Interface (I3C TCRI<sup>SM</sup>)*, Version 1.0,  
345 MIPI Alliance, Inc., 24 May 2022 (MIPI Board Adopted 7 September 2022).
- 346 [MIPI07] *Specification for I3C Basic<sup>SM</sup>*, Version 1.1.1 including Errata 01, MIPI Alliance, Inc.,  
347 9 June 2021 (MIPI Board Adopted 23 July 2021, Errata 01 dated 11 March 2022).
- 348 [MIPI08] *Frequently Asked Questions (FAQ) for MIPI I3C® v1.1.1 & MIPI I3C Basic<sup>SM</sup> v1.1.1*,  
349 FAQ version 1.1, MIPI Alliance, Inc., 22 August 2022  
(MIPI Board approved 22 August 2022).
- 351 [NXP01] UM10204, *I<sup>2</sup>C Bus Specification and User Manual*, Rev. 6, NXP Corporation N.V.,  
352 4 April 2014.
- 353 [PCISIG01] PCI SIG, *PCI Code and ID Assignment Specification*,  
354 <<https://pcisig.com/specifications>>, Revision 1.9, 31 May 2017.
- 355 [USB01] *Universal Serial Bus Specification*, <<https://www.usb.org/document-library/usb-20-specification>>, Revision 2.0 (including errata and ECNs through December 21, 2018),  
356 USB-IF, June 27, 2017.
- 358 [USB02] *Universal Serial Bus 3.2 Specification*, <<https://www.usb.org/document-library/usb-32-specification-released-september-22-2017-and-ecns>>, Revision 1.0 (including errata and  
359 ECNs through July 24, 2018), USB-IF, September 22, 2017.
- 361 [USB03] *Universal Serial Bus I3C Device Class Specification*, Revision 1.0, USB-IF,  
362 7 January 2022.

## 4 Technical Overview

The MIPI I3C Specification [[MIPI02](#)] defines the behavior of the Devices on an I3C Bus. This ensures Device compatibility and interoperability. However, hardware manufacturers and platform integrators have the challenge of designing and integrating I3C Bus Controller Logic as part of a subsystem that can connect to a Host through a system bus. To promote interoperability, MIPI has defined this I3C Host Controller Interface which allows for easier adoption of I3C into various ecosystems. This enables the development of common software driver implementations, while allowing for vendor-specific innovation.

The I3C Host Controller Interface (HCI) is a standard interface that allows for easy integration of I3C Bus Controller Logic, as an I3C Host Controller as a subsystem. This specification defines the high-level architecture of the I3C Host Controller subsystem, presented in [Figure 2](#).

An I3C Host Controller may be implemented using any system bus, however this specification specifically addresses the needs of PCIe Bus attached Host Controllers, AMBA Bus attached Host Controllers, and USB (Universal Serial Bus) attached Host Controllers. Specifics for system bus implementation can be found in [Section 9](#).



**Figure 2 I3C System Overview**

## 4.1 Scope

This Specification defines the software interface exposed by I3C Host Controller hardware, that allows for implementation of I3C Specification features in a flexible, extensible, and resource-efficient manner. The Host Controller Interface defines the necessary attributes and capabilities, and is based on the Transfer Command/Response Interface (TCRI, see MIPI I3C TCRI Specification [[MIPI06](#)]), which defines the expected hardware response and behavior for specific software actions. However, neither this I3C HCI specification nor the I3C TCRI Specification define the internal I3C Host Controller implementation. This specification defines the required steps of Host Controller programming, however additional steps might be required for specific Host Controller implementations, or for vendor-specific extensions.

This Specification does not define system-specific or platform-specific setup, I3C software architecture including compliance to available (I<sup>2</sup>C) solutions, overall platform power management requirements, or platform interrupt mechanisms. However, it does require certain capabilities that relate to Host software, interaction with Host System power management actions, and Host system support for receiving certain types of interrupts (if available to the System Bus).

This Specification also defines an optional Direct Memory Access (DMA) capability, which assumes that DMA engines are available within the I3C Host Controller subsystem, to allow the Host Controller direct access to Host system memory. These DMA engines use a memory access capability enabled by the system bus, that allows a Device to initiate such read/write requests (see [Section 9.1](#)). The use of integrated DMA engines enables a higher-performance operating mode (DMA Mode) for the Host Controller, which enables the Host Controller to initiate its own Host system memory requests without direct involvement from software and the notify the Host once enqueued transactions have been completed. In this manner, software may enqueue multiple I3C transactions that are not blocked by the sizes of the Host Controller's internal resources (i.e., FIFOs, queues, and buffers).

However, specific system bus implementations might not support such a memory access capability, which limits use of certain features defined in this Specification, including DMA Mode. For such system bus implementations, operation of the Host Controller is driven by directed reads/writes initiated by the Host and its software, to program the registers as well as internal resources (i.e., FIFOs, queues, and buffers) for all enqueued transfers. The Host Controller then fully relies on its internal memories, queues, and buffers to drive all transfers on the I3C Bus, and must be explicitly programmed with a high degree of direct involvement from software. Such an operating mode is referred to as PIO Mode.

Wherever possible, this Specification attempts to abstract the common elements, attributes, and flows used by the Host and its software to interact with the Host Controller, as well as those that introduce differences between PIO Mode and DMA Mode, as well as the use models for a Host Controller that might support one (or both) operating modes. The key commonality between PIO Mode and DMA Mode is the separation of the I3C Transfer Command/Response Interface (which handles Host-initiated transactions such as reads, writes, and CCCs) from the IBI interface (which handles Target-initiated Interrupt Requests of various types).

## 4.2 I3C HCI Purpose

The I3C Host Controller Interface (HCI) is intended to standardize the interface that platform software uses to access I3C Devices and capabilities. Naturally, per the I3C Specification, this Host Controller Interface supports I<sup>2</sup>C Devices, with compatibility as described in the I3C Specification. This Specification defines a standard interface/transport between platform software and I3C peripherals (such as sensor Devices). It does not limit the classes or capabilities delivered by any Devices enumerated on an I3C Bus, so long as all such Devices conform to the I3C Specification.

Implementing the I3C HCI Specification can eliminate requirements for platform vendors to develop and maintain their own Driver stacks. A standardized I3C HCI allows Operating System vendors (OSVs), developers, and distributors to offer software Drivers that are portable across hardware platforms.

### 4.3 I3C Key Features

The I3C HCI Specification provides efficient means for platform software to interface to the features provided by the I3C Bus, and ensures power-efficient operation of the Host Controller. The latter is specifically important as many I3C Device implementations typically target battery-powered environments.

In particular, the I3C HCI supports the following I3C Specification features:

- Two wire serial interface up to 12.5 MHz using Push-Pull with the following Data Rates supported:
  - I<sup>2</sup>C compliant Data Rates:
    - I<sup>2</sup>C Fast Mode (FM): 0 to 400Kb/s
    - I<sup>2</sup>C Fast Mode Plus (FM+): 0 to 1Mbps
  - Single Data Rate (SDR): I3C enhanced version of the I<sup>2</sup>C protocol, running up to 12.5 MHz:
    - I3C Coding SDR with Directed and Broadcast Common Command Codes (CCC)
  - Optional High Data Rate (HDR) Modes: Additional I3C Modes, not available for the I<sup>2</sup>C protocol Devices:
    - HDR-Dual Data Rate (HDR-DDR)
    - HDR-Ternary Symbol Legacy Mode (HDR-TSL)
    - HDR-Ternary Symbol for Pure Bus Mode (HDR-TSP)
    - Special end transfer handling for these HDR Modes (optional) including configuration of Targets with the [ENDXFER](#) CCC
- Supported Bus Roles:
  - I3C Primary Controller as the initial Active Controller
  - I3C Secondary Controller that acquires the Controller Role and becomes Active Controller (optional)
- Legacy I<sup>2</sup>C Target Device co-existence on the same Bus instance (with limitations as described in the I3C Specification)
- Legacy I<sup>2</sup>C messaging
- Multi-Drop capability
- Dynamic Addressing Assignment, namely:
  - Without pre-assigned Static Addresses (using the [ENTDAA](#) CCC)
  - With pre-assigned Static Addresses (using the [SETDASA](#) and [SETAASA](#) CCCs)
  - While supporting Static Addressing for Legacy I<sup>2</sup>C Target Devices
- In-Band Interrupt support
- Hot-Join support
- Asynchronous Time Stamping (optional)
- Common Command Code (CCC) Framing, namely:
  - CCCs (Direct and Broadcast) in SDR Mode only
- Target Reset without additional wires, including:
  - Target Reset Pattern, to Reset I3C Peripheral or Reset Whole Chip (i.e., Target)
  - Target Reset Pattern with the [RSTACT](#) CCC, for various directed reset actions
- Several other optional features in version 1.1.1+ of the I3C Specification [[MIPI05](#)], including:
  - Grouped Addressing (Groups)
  - Error Detection and Recovery Methods
  - Support for Monitoring Devices that use an early termination capability to end an ongoing data transfer (optional)

463 A Host Controller that conforms to this I3C HCI Specification includes the following high-level  
464 capabilities, features, and interface aspects, for Host software use:

- 465 • Direct data interface support (PIO Mode), with programmable buffer depths for the transmit/response and  
466 data buffers
- 467 • DMA interface support (DMA Mode):
- 468   • Supports Scatter-Gather transfers for data buffers
- 469 • Interrupt:
- 470   • Supports polling and interrupt controlled operation
- 471   • Selective interrupts can be masked
- 472   • Force Interrupt Register for software debugging
- 473 • Extensions:
- 474   • List of Extended Capability structures, including vendor-specific structures, to allow for more  
475 sophisticated hardware or additional functionality

#### 4.4 I3C HCI Fundamental Principles

476 The I3C HCI is designed to support the full functionality defined in the I3C Bus specification, as well as  
477 some addition features that are useful for modern platforms.

478 The fundamental principles for this specification are to:

- 479 • Enable standard Drivers to allow I3C Host Controller Driver support to be owned by Operating System  
480 vendors (OSVs), developers, and distributors
- 481 • Support capabilities of the I3C Specification, with focus on:
- 482   • Excellent power management capabilities
- 483   • Enable Deep Sleep states
- 484 • Define a flexible architecture that enables scalable capabilities at the discretion of the implementer, in  
485 order to reduce software involvement per transaction, such as:
- 486   • Optional capabilities and choice of operating modes, that allow for greater software control over  
487 incoming interrupt latency and threshold controls
- 488   • Auto-Command capabilities and DMA Mode, which can maximize time between software wake-ups,  
489 and enable more efficient processing of some I3C Bus events, entirely within the Host Controller  
490 subsystem
- 491 • Provide a well-defined, comprehensive software interface with full support for I3C Bus Initialization,  
492 Bus Configuration and all transaction types in supported I3C Modes
- 493 • Design to work with a variety of system buses, such as SoC fabrics
- 494 • Provide Device enumeration flows
- 495 • Provide Device operation flows, including power management aspects
- 496 • Define PIO (Programmable I/O) Mode transfer flows
- 497 • Define DMA (Direct Memory Access) Mode transfer flows
- 498 • Reuse register definitions between PIO Mode and DMA Mode, where possible
- 499 • Use a Command/Response flow that defines various data structure formats (called Descriptors) for  
500 transaction flows:
- 501   • Allows for simple transfers, but also enables more complex multi-part transactions
- 502   • Supports selected optional HDR Modes and provides a path for future extensibility, as the I3C  
503     Specification evolves
- 504 • Define DMA Scatter-Gather support for data blocks (i.e., no buffer copy operation)

- 505 • Minimize Register Map accesses:  
506   • None when idle  
507   • Minimum Register Map transactions required to set up a Command and schedule transfers  
508   • In DMA Mode, the Command Completion status available in Driver allocated memory to interrupt  
509    handlers  
510 • Enable support for essential basic features of the I3C Specification version 1.1 [**MIPI02**]

511 **Note:**

512   *This version of the I3C HCI Specification does not support all optional features and capabilities that  
513   are supported in version 1.1+ of the I3C Specification [**MIPI05**]. A future version of the I3C HCI  
514   Specification is expected to add support for these optional features and capabilities.*

515 This Specification is written to cover the following capabilities:

- 516 • Transaction capability:  
517   • Schedule back to back DAA (Device Address Assignment) commands to offload Device Enumeration  
518    process to Host Controller hardware  
519   • Capability to communicate to the same Device at multiple supported clock speeds, on a per-transaction  
520    basis  
521   • Capability to send Broadcast CCC and Direct CCC transfers using managed CCC transfer framing:  
522     • Includes support for all variants of Direct CCC transfers, such as Direct Write/SET, Direct  
523       Read/GET, and any valid combinations of the above  
524     • May address one or multiple Target Devices using Dynamic Addresses; may also send to Group  
525       Addresses (Direct Write/SET only)  
526   • Capability to support In-Band Interrupts (IBIs), with or without Mandatory Data Byte (MDB) and  
527    optional data payload  
528   • Capability to configure auto-accept or auto-reject for Target-initiated interrupt requests, such as In-  
529    Band Interrupts, Hot-Join Requests, and Controller Role Requests:  
530     • May enable per-Target configuration using DAT entries  
531     • May enable automatic NACK and directed **DISEC** CCC to disable some Target-initiated interrupt  
532       requests  
533   • Support pure-I3C and mixed Bus configurations  
534   • Address optimization, Abort and Resume capabilities  
535 • PIO Mode operation for light designs:  
536   • Programmable Thresholds to generate I3C transactions minimizing idle times on Bus  
537 • DMA Mode with Command Rings, to enable clean Doorbell mechanisms:  
538   • Multiple Command/Response Rings and IBI Rings, including IBI payload  
539   • Interrupt aggregation / coalescing  
540 • Software discoverable capabilities

## 4.5 I3C HCI Relationship to Other MIPI Specifications

541 This Specification describes the set of required and optional behaviors for an I3C Host Controller that  
542 contains one or more instances of I3C Bus Controller Logic, where each instance can drive transactions on  
543 its connected I3C Bus.

544 The I3C Bus Controller Logic shall comply with version 1.1.1+ of the MIPI I3C Specification [**MIPI05**] as  
545 an I3C Controller. The I3C Specification defines the protocol details and electrical requirements for the I3C  
546 Bus Controller Logic that acts an I3C Device on an I3C Bus.

547 The I3C Bus Controller Logic also uses the I3C Transfer Command/Response Interface, specified in  
548 [**MIPI06**], for its normative definition of Transfer Commands and Transfer Responses. The MIPI I3C TCRI

549 Specification defines the operational details of how the I3C Bus Controller Logic processes such Transfer  
550 Commands, drives I3C transactions on the I3C Bus, and generates Transfer Responses.

551 An instance of an I3C Host Controller can optionally be discoverable to system software using a platform  
552 resource that describes its capabilities. This platform resource can be an ACPI-compliant structure that  
553 includes properties defined by the MIPI DisCo<sup>SM</sup> Specification for I3C [*MIPI04*]. The DisCo<sup>SM</sup>  
554 Specification for I3C Specification defines a set of properties and their hierarchy within an ACPI structure,  
555 in order to describe the I3C Bus, its I3C Bus Controller instance, and any attached Target Devices.

## 4.6 What's New in I3C HCI Version 1.2

556 This version of the I3C HCI Specification introduces important changes to existing features defined in  
557 version 1.1, as well as several new features and capabilities.

558 Most of the new features and capabilities are Optional-Normative, meaning that they are optional for the  
559 implementer to either support or not support, and that the definition is considered to be normative, as  
560 defined in this I3C HCI Specification. Implementers may choose to include any of these features and  
561 capabilities, in order to serve various use cases. Some of these features and capabilities are intended to  
562 enhance and extend the I3C Device roles that are possible for Host Controller implementations, and others  
563 provide extended functionality to offload Host processing for certain I3C Bus transfers with Target  
564 Devices. Since these Optional-Normative features and capabilities are not required for all Host Controller  
565 implementations, implementers are not required to include any of them in a particular implementation;  
566 however, if included, implementers should regard the relevant sections of this Specification as normative  
567 for those new capabilities and features.

568 Additionally, this version of the I3C HCI Specification uses the I3C TCRI Specification [*MIPI06*] as the  
569 normative reference for interacting with the I3C Bus Controller Logic within the Host Controller.

570 The key changes in I3C HCI v1.2 are:

- 571 • All issues addressed by Errata 01 for I3C HCI version 1.1
- 572 • Optional Scheduled Command processing that allows the I3C Bus Controller Logic to periodically  
573 initiate Transfer Commands (see *Section 6.16*):
  - 574 • Reduces software overhead by giving the Host Controller autonomous capability to drive repeatable  
575 Transfer Commands for I3C Targets according to a configured schedule
  - 576 • Defines several possible handler options for Scheduled Commands configuration and processing
- 577 • Optional Standby Controller mode with Secondary Controller Logic (see *Section 6.17*)
- 578 • Optional Dead Bus Recovery Mechanism that allows the Host Controller to test the I3C Bus to determine  
579 whether an Active Controller is present, and choose its initial role accordingly (see *Section 6.18*)
- 580 • Improvements to In-Band Interrupt (IBI) status reporting for different events (see *Section 8.6*):
  - 581 • Changes to fields in IBI Status Descriptor to allow more detailed reporting of Auto-Command  
582 responses
  - 583 • Added support for reporting status of Scheduled Command processing
- 584 • Optional I3C Target credit counting mechanism to help the Host manage whether Target IBI Requests  
585 will be accepted based on activity level and readiness (see *Section 6.9.5*)
- 586 • Support for Host-initiated abort of IBI data payloads:
  - 587 • Allows Host Controller to continue sending IBI data payload chunks to the Host, until the Host  
588 indicates when the Bus Controller Logic should end the transfer from the Target (See *Section 6.9.4* and  
589 *Section 7.4.18*)
- 590 • Improvements to PIO Mode capabilities:
  - 591 • Added optional support for the Response Queue size to be defined separately from the Command  
592 Queue size (see *Section 6.5* and *Section 7.5.8*)
  - 593 • Added optional support for extended IBI Queue sizes, for implementations that will support more  
594 frequent use of IBI Requests on the I3C Bus (see *Section 7.5.8*)
  - 595 • More flexible control over the operational status of the PIO Queues (see *Section 7.5.13*)

- 596     • Support for HDR Mode Data Transfer and Early Termination capabilities for optional HDR Modes:  
597         • Improves robustness of I3C Bus transfers, and aids adoption of I3C version 1.1.1+ for various use  
598         cases  
599         • Available for HDR-DDR Mode and HDR-Ternary Modes (see *Section 6.8.6* and *Section 8.4.2.6*)  
600     • Document structure/scope changes and content restructuring:  
601         • Normative definitions of Command Descriptors (i.e., Transfer Commands), Response Descriptors, and  
602         flows for processing Transfer Commands now appear in the separate I3C Transfer Command/Response  
603         Interface Specification (see [**MIPI06J**]), not in this I3C HCI Specification

This page intentionally left blank.

## 5 Architectural Overview (informative)

### 5.1 Interface Architecture

The I3C Host Controller is envisioned as hardware that supports I3C Controller-capable Device functionality, and might also support I3C Target (i.e., I3C Secondary Controller) functionality or other vendor-defined functionality with the use of vendor-specific Extended Capabilities (see [Section 5.5](#)). However, the primary use case is optimized to support the I3C Active Controller role, and intended to enable the Host Controller as the Primary Controller of the I3C Bus.

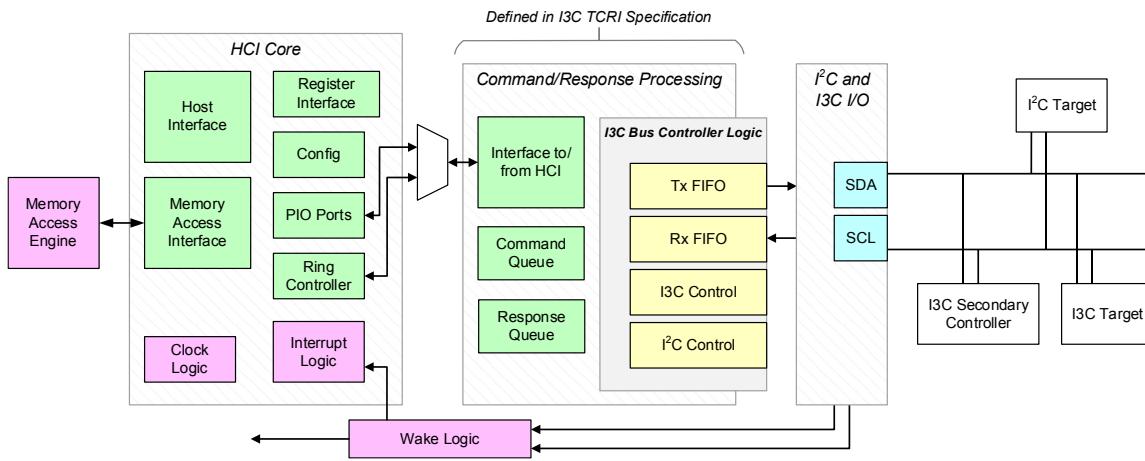
The Host interacts with the Host Controller using a standardized Register Map (see [Section 7.3](#)) which allows software (i.e., a Driver) to initialize and configure the Host Controller, discover its capabilities, enumerate and detect I3C Target Devices on the I3C Bus, and drive transfers on the I3C Bus using the Transfer Command/Response Interface, while operating as Active Controller. The Register Map is divided into sections to allow for flexibility of implementation as well as extensibility.

If supported by the System Bus (per [Section 9](#)), a Host Controller can optionally support DMA capability, which enables direct access to Host system memory. Such a Host Controller uses its DMA engines to support DMA Mode, a higher-performance operating mode that allows the Host Controller to drive Transfer Commands more efficiently. A Host Controller might also support PIO Mode, a register-based operating mode that does not require DMA engines, and requires software to manage all transfers via register operations.

If the DMA engines used to transfer data to/from the Host Controller are implemented with DMA capability, then they are implementation-specific and are not defined by this Specification, with one exception: if the system bus supports 64-bit-addressable memory access, then the DMA engines will support 64-bit addressing. The DMA engines are used by the Memory Access Interface logic (see [Figure 3](#)) to directly request reads or writes of Host System memory for specific purposes as defined in this Specification, without any direct software involvement.

If any configuration of DMA engines is required, then implementers can use a proprietary mechanism, preferably using a vendor-specific Extended Capability structure. The DMA engines are applicable only for a system bus that supports direct access to Driver-allocated Host memory from the Device (i.e., DMA per [Section 9.1](#)).

[Figure 3](#) shows a high-level example of a Host Controller implementation. In this particular example the Memory Access Engine (e.g., the DMA engine) is external to the IP, even though the Memory Access Engine and the Host Interface block can both access the Host system bus. The I/O block is also not a part of the Host Controller programming. Any I/O-specific programming is beyond the scope of this Specification, and if required could be implemented through a vendor-specific Extended Capability structure. Also, the presence and implementation of the Wake Logic function is application-specific and beyond the scope of this I3C HCI Specification. In this example both PIO Mode and DMA Mode are supported, although specific implementations are not required to support both modes. Additionally, a MUX represents the mechanism for selecting the operating mode (i.e., PIO vs. DMA) where the I3C Bus Controller Logic interacts with either the PIO Queue Port registers or the Ring Controller (i.e., the Command/Response Ring Pair for the active Ring Bundle).

**Figure 3 Example Host Controller Implementation****Note:**

In **Figure 3** above, a MUX represents the mechanism for selecting the operating context. This mechanism allows the I3C Bus Controller Logic to process Command Descriptors and generate Response Descriptors based on the current operating mode (i.e., PIO vs. DMA). A Host Controller could optionally have other logic that adds other operating contexts (e.g., Scheduled Commands logic, per **Section 6.16**) that can be selected based on configuration as well as current transfer activity.

In implementations like the one shown in **Figure 3**, the I3C Bus Controller Logic box generally contains blocks for Tx FIFO, Rx FIFO, I3C Control, and I<sup>2</sup>C Control, per the I3C TCRI Specification (see [**MIPI06**] at **Section 5.1**). The Transfer Command/Response Interface is used by the I3C Bus Controller Logic for processing Transfer Commands received from the Host (with optional TX data), and for returning any resulting Transfer Responses to the Host (with optional TX data).

From an internal architecture perspective, the Host Controller uses these blocks as follows (see **Section 5.3** for full details regarding the Command Interface):

- If the Host Controller supports PIO Mode:** The Host Controller processes transactions that the Host previously enqueued via writes to the PIO Mode Registers section of the Register Set. The Host Controller might not use (or require) a Memory Access Interface. This requires careful coordination of the manner in which the Host writes data to the Tx FIFO (i.e., the TX Data Buffer) to ensure that large Write transactions do not overflow the free space in this Tx FIFO, or cause it to underflow during a transaction. Similar coordination is required for Read transactions, to ensure that the Host reads incoming data from the Rx FIFO (i.e., the RX Data Buffer) such that it does not overflow and force the transaction to be cancelled. See **Section 6.5** for more details on transfers using PIO Queues.
- If the Host Controller supports DMA Mode:** The Host Controller processes transactions that were previously written into the Ring's cyclic buffer. Writes are indicated by a difference between the values of the internally maintained Dequeue-Pointer vs. the Host-updated Enqueue-Pointer. The Memory Access Interface then requests the necessary data structures from Host system memory (the Transaction Descriptor of each transaction entry enqueued in the Ring includes a memory pointer), manages the process of writing data into the Tx FIFO, and drives the transaction to the I3C Bus. If the transaction reads data from a Target Device, then the Memory Access Interface also manages the process of reading response data from the Rx FIFO and writing it into Host system memory, at the appropriate location. See **Section 5.4** and **Section 6.6** for more details on transfers using Rings.

## 5.2 General Information

The Host Controller supports Active Controller mode, which enables a Command/Response interface that the Host uses to enqueue transfers and read response status, while the Host Controller's I3C Bus Controller Logic acts as the Active Controller of the I3C Bus. If optional Secondary Controller support is implemented, then the Host Controller also supports Standby Controller mode (per [Section 6.17](#)).

The Host Controller supports up to 32 I3C/I<sup>2</sup>C Devices on a single I3C Bus instance. The actual number of Devices supported by an implementation might depend on the implemented size of the Device Address Table (DAT). This size is software-discoverable.

To perform the transfer or CCC, the Host Controller can support either the PIO (register access) method, or Rings dedicated to DMA engines, or both:

- **When using PIO Mode:** A single Command Port, Response Port, and IBI Port are available to software.
  - The Command Port and Response Port use the Command Queue and Response Queue in the I3C TCRI instance, per the I3C TCRI Specification (see [[MIPI06](#)] at [Section 6.2](#)).
  - The IBI Port uses the IBI Queue, which stores IBI Status Descriptors (see [Section 8.6](#)) and IBI Data DWORDs in sequence.
- **When using DMA Mode:** A Host Controller supports up to eight Ring Bundles. Each instance has a Ring Header with registers; a Command/Response Ring Pair; and an IBI Ring Pair. These Ring Bundles are contained within the Ring Controller logic, see [Section 5.1](#).
  - The active Ring Bundle's Command/Response Ring Pair uses the Command Queue and Response Queue in the I3C TCRI instance, per the I3C TCRI Specification (see [[MIPI06](#)] at [Section 6.2](#)).
  - The IBI Status and Data Ring Pairs use the IBI Queue, with IBI Status Descriptors (see [Section 8.6](#)) and IBI Data DWORDs (respectively). IBIs from individual Targets can be steered to any valid IBI Ring Pair, per the Ring Bundle ID in the Target's DAT entry.
  - If the Host Controller supports multiple Ring Bundles, then the Ring Controller is responsible for choosing the next Ring Bundle to execute Commands/Responses, such that only one Ring Bundle is active at any time. Any inactive Ring Bundles are held in a static state until the Ring Controller arbitrates and chooses another to be active.

**Note:**

*The actual number of Ring Bundles supported in an implementation can be discovered via registers in the Ring Header section.*

[Table 1](#) lists Host Controller Parameters and their limitations. Software can optionally set limits on the usage of particular Host Controller features within the provided Host Controller capabilities.

704

**Table 1 Host Controller Parameters**

Parameter	Description	Hardware Capability (max)	Limited with Software (min, max)	Notes
<b>DEVICES</b>	Number of Devices supported	32	(1, 32)	The number of Devices supported by Host Controller implementation might be limited by hardware, as indicated by fields <b>TABLE_OFFSET</b> and <b>TABLE_SIZE</b> in register <b>DAT_SECTION_OFFSET</b> ( <a href="#">Section 7.4.11</a> ). The Host Controller only uses table entries with a valid Dynamic Address. Entries with the value 7'h00 are not used. In some implementations, this parameter only applies to Devices expected to send Interrupt Requests, for which the Host Controller configures auto-accept or auto-reject based on the Dynamic Address.
<b>BUSES</b>	Number of I3C Buses supported	15	(No software configurable limits)	This setting depends upon the particular system bus. One I3C Bus instance is supported with one instance of I3C Bus Controller Logic. Specific hardware implementations might implement multiple I3C Bus Controller instances. If permitted for the system bus, then additional Bus Controller instances are possible, using the Multi-Bus Instance Extended Capability structure ( <a href="#">Section 7.7.4</a> ) in the Extended Capabilities list.
<b>PTRS_PIO</b>	Number of queue instances (ports) supported in PIO Mode	One set of 4 queues: Command, Response, Data, IBI	(No software configurable limits)  Available only while operating in PIO Mode	All four queues/ports in the set comprise a <b>single interface</b> to I3C Bus Controller Logic (i.e., one HC Interface with one instance of Bus Controller Logic, as defined in the I3C TCRI Specification [ <a href="#">MIPI06</a> ]). Presence is indicated by register <b>PIO_SECTION_OFFSET</b> ( <a href="#">Section 7.4.14</a> ). Queues/ports are accessed via registers in the PIO Mode Registers section ( <a href="#">Section 7.5</a> ). Command and Response use dedicated Queue Ports. The transfer payload data is stored and fetched from single Data Port. IBI Port features IBI Status Descriptor structure and the IBI payload data.
<b>RNGS_DMA</b>	Number of Ring instances supported in DMA Mode	8	(0, 8)  Available only while operating in DMA Mode	Presence of Ring Headers is indicated by register <b>RING_HEADERS_SECTION_OFFSET</b> ( <a href="#">Section 7.4.13</a> ). Ring Headers are accessed by registers in the Ring Headers Specific Registers section ( <a href="#">Section 7.6</a> ). Hardware exposes the number of Ring Headers via field <b>MAX_HEADER_COUNT_CAPABILITY</b> in register <b>RHS_CONTROL</b> ( <a href="#">Section 7.6.1</a> ). Software can limit this, using field <b>MAX_HEADER_COUNT</b> in the same register.

### 5.3 Command Interface

The Transfer Command/Response Interface (TCRI) provides access to the I3C Bus Controller Logic in two operating modes: Programmable I/O (PIO) Mode, and Direct Memory Access (DMA) Mode. A Host Controller supports at least one of these modes, and might support both of them, at the discretion of the implementer.

If **PIO Mode is supported**, then the following capabilities, attributes, and logic must be implemented:

- The PIO Mode Registers section (*Section 7.5*)
- A valid (non-zero) value in the **SECTION\_OFFSET** field of register **PIO\_SECTION\_OFFSET** (*Section 7.4.14*), pointing to the starting offset of the PIO Mode Registers section
- Support for PIO Interrupt logic that allows PIO-specific interrupts to be routed to the Host via the system bus

If **DMA Mode is supported**, then the following capabilities, attributes, and logic must be implemented:

- Ring Controller logic, including one or more Ring Bundles each exposing its own set of Ring Header registers
- In the Ring Headers Specific Registers section (*Section 7.6*), from 1 to 8 Ring Header offsets, one per supported Ring Header (i.e., Ring Bundle) within the Ring Controller logic
- Memory Access Interface that uses DMA engines (i.e., Memory Access Engines per *Figure 3*) to access Host System memory
- A valid (non-zero) value in the **SECTION\_OFFSET** field of register **RING\_HEADERS\_SECTION\_OFFSET** (*Section 7.4.13*), pointing to the starting offset of the Ring Headers Specific Registers section
- Support for Ring Bundle Interrupt logic that allows Ring-specific interrupts to be routed to the Host via the system bus

Both operating modes (i.e., PIO Mode and DMA Mode) are regarded as “transports” that enable the software to schedule Transfer Commands on the I3C Bus. While the details of the transport layer differ significantly per operating mode, the meaning of the Transfer Commands (i.e., special Command Descriptors that describe I3C transfers) that are enqueued, as well as the responses that might be generated for such Transfer Commands (i.e., Response Descriptors), are fundamentally the same for both operating modes. Additionally, the software enqueues Transfer Commands in the current operating mode, individually or in ordered sequences (per *Section 6.12*). Once Transfer Commands are enqueued, the I3C Bus Controller Logic eventually processes them, as defined in the I3C TCRI Specification (see [*MIPI06*] at *Section 6.2*).

PIO Mode and DMA Mode are mutually exclusive operating modes, i.e., only one of them can be enabled at the same time. As a result, the Host Controller will prevent PIO Mode and DMA Mode from both being enabled simultaneously. Software selects the current operating mode transfer processing by writing to field **MODE\_SELECTOR** in register **HC\_CONTROL**.

**Note:**

Field **MODE\_SELECTOR** is a read-only field while the Host Controller is in the **ENABLED** state, and writes to this field will have no effect. To switch modes, the software must first disable the Host Controller by clearing the **BUS\_ENABLE** field (in the same register).

In v1.0 of I3C HCI (this specification), the current operating mode was implied by the number of Ring Headers in **ENABLED** state. Software would switch operating modes by either (1) If all Ring Headers were disabled, enabling one Ring Header to switch from PIO Mode to DMA Mode, or (2) If only one Ring Header were enabled, disabling it to switch from DMA Mode to PIO Mode. Starting with v1.1 of the I3C HCI specification, field **MODE\_SELECTOR** replaces that mechanism, providing direct control instead.

Software developers are advised to note this change and ensure that any software Drivers used with Host Controllers conforming to I3C HCI v1.1 or newer use the **MODE\_SELECTOR** field to explicitly switch operating modes.

752 The transaction interface to the Host software differs significantly between PIO Mode and DMA Mode:

- 753 • **PIO Mode:** In PIO Mode, the Host Controller exposes Command, Response, Rx Data, Tx Data, and IBI  
754 Queues via a register interface (the Queue Port registers; see *Section 6.5* and *Section 7.5*). The queue  
755 operation assumes that a write to the Command Queue will result in operation on the Bus, for a valid  
756 Transfer Command. Similarly, a read out from the Response Queue indicates to the Host Controller that  
757 the software has consumed the Transfer Response for a previously written Transfer Command.

758 In PIO Mode, the software does not allocate memory for Command, Response, Data, and IBI  
759 Buffers; instead, the Host Controller directly writes to registers to drive transactions, and maintains  
760 transaction status for its use of all Host Controller internal memories (i.e., queues/buffers) in order to  
761 avoid overflow/underflow situations. The Host Controller's PIO Queues have fixed sizes that are  
762 determined by the implementer, and software sets the thresholds for queue notifications and other  
763 activities to suit the use case.

- 764 • **DMA Mode:** In DMA Mode, the Host Controller exposes one or more Ring Headers (see *Section 7.6.10*).  
765 Each Ring Header defines a Command/Response Ring Pair (Command Ring and Response Ring, see  
766 *Section 6.6.2*), and an IBI Ring Pair (IBI Status and IBI Data Ring, see *Section 6.6.3*). Software  
767 determines whether to initialize the IBI Ring Pair, depending upon the use case: if so, then software  
768 allocates and provides Driver-allocated memory for Rings. The Driver can schedule the bulk of  
769 transfers/commands on the Ring, and notify the Host Controller after the entire bulk is ready. Compared  
770 to PIO Mode, DMA Mode gives more flexibility in terms of the time at which the transfer occurs on the  
771 I3C Bus. Similarly for Response processing, interrupt coalescing can be used to improve power  
772 efficiency.

773 For DMA Mode, the Transfer Descriptor structure (see *Section 8.3*) is used for scheduling the transfer or  
774 command. The Transfer Descriptor structure incorporates the Command Descriptor structure (see  
775 *Section 8.4*). Similarly, the Response Descriptor structure (see *Section 8.5*) is used for reception of transfer  
776 or command status.

777

**Table 2 Queues and Rings Available in PIO and DMA Modes**

Queue/Ring	PIO Mode	DMA Mode	Interface Method (Descriptor or Register Used)		Notes
			PIO Mode	DMA Mode	
Command	Queue	Ring	Command Descriptor	Transfer Descriptor, featuring Command Descriptor	In DMA Mode, the Transfer Descriptor defines pointer(s) to data buffer(s) where Raw Data is stored to (write data) or fetched from (read data)
Response	Queue	Ring	Response Descriptor	Response Descriptor	–
Rx Data	Queue	n/a	RX Data Port (Register)	Data Buffer Pointer(s) in Transfer Descriptor	In PIO Mode, the Data Queue Port is used to fetch transfer read data
Tx Data	Queue	n/a	TX Data Port (Register)	Data Buffer Pointer(s) in Transfer Descriptor	In PIO Mode, the Data Queue Port is used to store transfer write data
IBI Status	Queue	Ring	IBI Status Descriptor via IBI Port (Register)	IBI Status Descriptor	In PIO Mode, a single queue is used for both IBI Status Descriptor and IBI Data (see <b>Section 6.9.1</b> ) In DMA Mode, each Ring Bundle supports IBI transfers if it is initialized. DAT table entries allow IBIs from specific Target Devices to be sent to a specific Ring Bundle.
IBI Data	Queue	Ring	(see Notes)	Raw Data	

778

**Note:**

*In Standby Controller mode (i.e., when this Host Controller does not currently hold the Active Controller Role), the IBI Queue (for PIO Mode) and IBI Rings (for DMA Mode) are also used to capture certain Broadcast CCC messages that the Active Controller on the I3C Bus might send. See **Section 6.17.3.2** for more details.*

779  
780  
781  
782

## 5.4 Rings Overview (DMA Mode)

A Host Controller can optionally support DMA operating mode for transfers. DMA Mode includes support for Rings, which are implemented within the Ring Controller logic, as described by the Ring Headers registers. The Ring Controller logic has the ability to initiate requests via the Memory Access Interface (as shown in *Figure 3*), concurrently for one or more Ring Bundles, as configured by the corresponding Ring Headers. The DMA engines manage these requests to the system bus as transfers to/from Host system memory for various data structures describing a Command/Response transfer, or an IBI transfer. The Ring Controller logic manages and buffers these transfers, in order to autonomously drive enqueued transactions using the I3C Bus Controller Logic, according to the readiness of data that must be written to the Tx FIFO. For response data, the Ring Controller and Memory Access Interface work together to ensure that data read from the Rx FIFO is written via the DMA engines to Host system memory.

In cases where access to Host system memory is delayed, and the Memory Access Interface does not receive responses to read/write requests in a timely manner, the Ring Controller optionally instructs the Controller Logic to stall the clock, according to the rules for Controller Clock Stalling in SDR Mode (see *Section 5.1.2.5* of the I3C Specification [*MIPI02*]). Or, in some cases, the Ring Controller must temporarily delay transaction processing for that particular Ring Bundle, and attempt to process enqueued transactions for another Ring Bundle that is enabled and running. The Ring Controller and Memory Access Interface will attempt to avoid situations where transactions must be cancelled or terminated early due to delays. In certain cases involving transactions in HDR Modes, the Ring Controller might need to exit an HDR Mode (and then re-enter the HDR Mode again at a later time) if the necessary memory cannot be read in time to initiate a new transaction, while remaining in the same HDR Mode as a previous transaction. If conditions warrant, then the Host Controller inserts these interruption events to replace the HDR Restart Pattern in HDR framing when a delay seems likely to disrupt the next enqueued transfer within the same HDR Mode.

The Host Controller exposes the DMA Mode capability via the presence of the Ring Headers Specific Registers section as defined in *Section 7.6* (i.e., a non-zero value in the **SECTION\_OFFSET** field of register **RING\_HEADERS\_SECTION\_OFFSET**, see *Section 7.4.13*). If the Ring Headers Specific Registers section is present, then DMA Mode is available and the Host Controller uses DMA Mode as its current operating mode, if configured by field **MODE\_SELECTOR** in register **HC\_CONTROL** (*Section 5.3*).

If PIO Mode is also supported, then software is allowed to switch the current operating mode by writing to this same field (**MODE\_SELECTOR** in register **HC\_CONTROL**) while the Host Controller is in **DISABLED** state. Software ensures that all Rings are fully processed and empty of outstanding transactions, and all Ring Headers are in **DISABLED** state, before switching the current operating mode.

Every Ring Header defines one Ring Bundle consisting of the following Ring Pairs:

- The first Ring Pair (Command/Response Ring Pair) supports regular, Controller-originated, transfers:
  - **Command Ring** to contain transfer information
  - **Response Ring** to contain transfer status

When the Ring Bundle is active, both of these Rings will provide access to the Command Queue and Response Queue within the I3C Bus Controller Logic via the TCRI, such that the enqueued Transfer Commands are processed and Transfer Responses are generated.

- Optionally, the second Ring Pair (IBI Ring Pair) supports IBI transfers:
  - **IBI Status Ring** to contain status for IBI transfers
  - **IBI Data Ring** to contain data for IBI transfers

In DMA Mode, the Host Controller supports up to eight such Ring Bundles. Software can optionally limit the number of Ring Bundles that are enabled for use, by writing to register **RHS\_CONTROL** (*Section 7.6.1*).

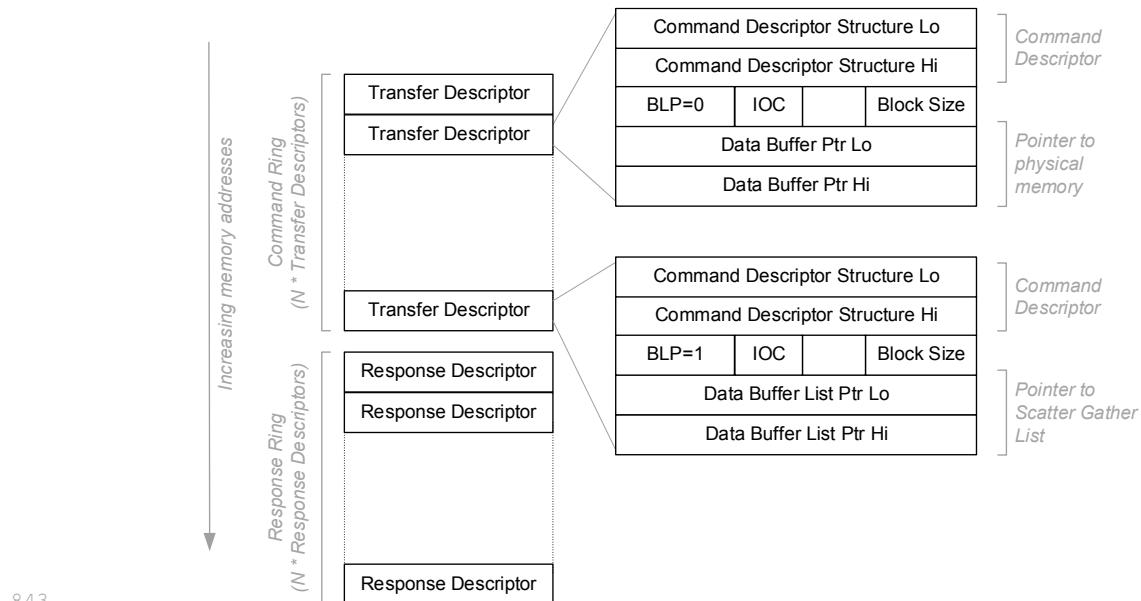
The IBI Ring Pair is optional for a given Ring Bundle. The option to not configure the IBI Ring Pair(s) for use might be applicable to setups with simple Devices (i.e., not producing IBI, Hot-Join, and Controller Role Request). But at least one IBI Ring Pair in each enabled Ring Bundle is typically allocated and configured for use, in order to receive In-Band Interrupt Requests for typical I3C Bus integrations. Additionally, the IBI Ring Pair for Ring Bundle 0 must be allocated and configured for use, in order to receive Hot-Join Requests.

In DMA Mode the data is provided either via a single Data Buffer (i.e., physically contiguous memory), or, for Scatter-Gather transfers, via a list of Data Buffers (i.e., a Buffer List Pointer).

**Figure 4** shows the Command/Response Ring Pair within a Ring Bundle. This figure shows how the Transfer Descriptors are arranged as data structures in memory. Each Transfer Descriptor contains the Command Descriptor. In this example, the Transfer Descriptor is shown with a single Data Buffer (i.e., a single region of physically contiguous memory) as well as a Data Buffer List (i.e., non-contiguous memory using Scatter-Gather). For additional details, see **Section 8.3**.

**Note:**

*The Rings defined in this Specification are single-segment only, and as a result can be presented as tables.*



**Figure 4 Example of Rings Used for Host-Initiated Transfers**

In addition, the IBI Rings handle all Target-initiated interrupt requests (i.e., In-Band Interrupts, Hot-Join requests, and Controller Role Requests) as well as (for Standby Controller mode only) data for any Broadcast CCCs that the Active Controller might send. The IBI Status Descriptor structure contains a Data Length field, which is used to advance the pointer in the Data part of the IBI Ring. The Status and Data parts of the IBI Ring might each have different (and possibly non-contiguous) physical memory locations, depending on the Scatter-Gather capabilities of the Host Controller (see **Section 6.2.1**). The IBI Status Ring and the IBI Data Ring will each occupy separate regions of memory.

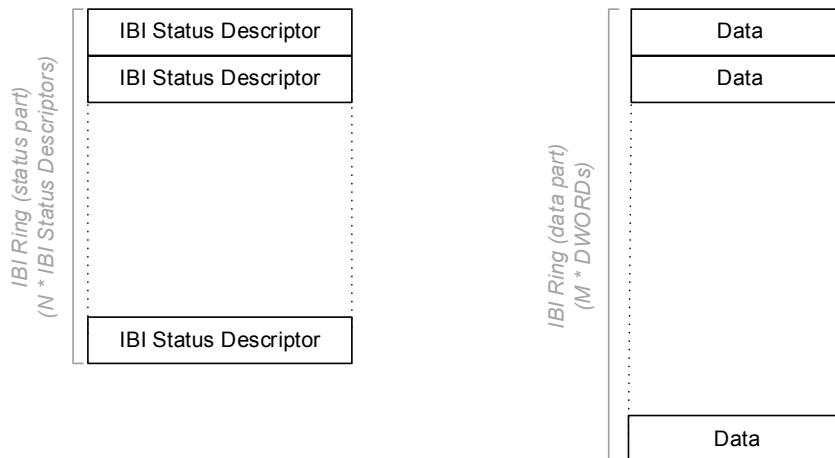
851 The Driver can optionally choose to not use the IBI Ring Pair for any given Ring Bundle (i.e., to use only  
 852 the Command/Response Ring Pair in that Ring Bundle). In this case all functionality related to IBI  
 853 Status/Data Ring Pair is disabled for that Ring Bundle. If no Ring Bundles have the IBI Ring Pair  
 854 configured for use, then any incoming requests will not be received, since the I3C Bus Controller Logic  
 855 must automatically NACK the IBI and also disable interrupts on the I3C Bus (i.e., by sending the **DISEC**  
 856 Broadcast CCC).

- 857 • For Active Controller mode, at least one Ring Bundle must have its IBI Ring Pair configured for use, in  
 858 order to receive any Target-initiated interrupt requests (i.e., In-Band Interrupts, Hot-Join requests, and  
 859 Controller Role Requests).
  - 860 • If multiple Ring Bundles are supported, and have their IBI Ring Pairs enabled, then IBI Requests from  
 861 a specific I3C Target Device will be steered to a specific Ring Bundle ID, using the **RING\_ID** field in  
 862 the DAT entry for a Device's Dynamic Address (see **Section 6.9.2**).
  - 863 • Hot-Join requests will be steered to Ring Bundle 0, and any IBI Requests from other I3C Target  
 864 Devices that do not have a dedicated DAT entry will be steered to Ring Bundle 0 (see **Section 6.9.2**). If  
 865 Ring Bundle 0's IBI Ring Pair is not enabled, then any such requests will not be received.
- 866 • If the Host Controller also supports the optional Standby Controller mode (see **Section 6.17.3**), then in  
 867 order to receive notifications of certain Broadcast CCCs that might be sent by the Active Controller, at  
 868 least one Ring Bundle must have its IBI Ring Pair configured for use and enabled in Standby Controller  
 869 mode.
  - 870 • Per **Section 6.17.3.2**, the Ring Bundle ID indicated by field **BCAST\_CCC\_IBI\_RING** in register  
**STBY\_CR\_CONTROL** determines which IBI Ring Pair will receive these Broadcast CCCs. If that Ring  
 Bundle's IBI Ring Pair is not enabled, then any such Broadcast CCCs will not be received while the  
 Host Controller is operating in Standby Controller mode.

874 **Note:**

875 *Since the IBI Ring Pair for any given Ring Bundle is optional, there is a possibility that it might not  
 876 be configured for use. If the IBI Ring Pair is not configured for use, then both its IBI Status Ring  
 877 Base Address and its IBI Data Ring Base Address will not contain a valid address, and in its  
 878 **IBI\_SETUP** register (**Section 7.6.10.2**) the fields **IBI\_STATUS\_RING\_SIZE** and **CHUNK\_COUNT** will be  
 879 set to 0.*

880 **Figure 5** shows the IBI Ring Pair within a Ring Bundle.



881 **Figure 5 Rings Used for IBI**

882 If the IBI Data Ring does not have enough space to accept all of the payload Data received in an IBI,  
 883 including that from an Auto-Command, then the Host Controller either terminates reception of the IBI  
 884 Data, or uses clock Stalling:

- 885     • **If terminated or aborted:** The Host Controller reports only received data in the IBI Status Descriptor  
886       structure, and marks the **LAST\_STATUS** to be 1. In this situation the software would either re-allocate IBI  
887       Data Buffer memory to a larger size, or prioritize the thread servicing the IBI.  
888     • **If clock Stalled:** The Host Controller asserts an **IBI\_READY\_STAT** interrupt to indicate data/status  
889       availability in the IBI Ring, and the Driver can handle the partial IBI payload, since the **LAST\_STATUS**  
890       flag in the IBI Status Ring will not be set to 1. Note that the Driver must consider the IBI's data payload  
891       to be partial, until the Host Controller is able to provide a subsequent IBI Status Descriptor structure with  
892       **LAST\_STATUS** set to 1.

## 5.5 Extended Capabilities

893     The Extended Capabilities registers add functionality beyond the mandatory Controller-capable Device  
894       support, or enhance the implementation (e.g., better hardware identification).

895     The Extended Capability Pointer is available at a known location in the Capabilities and Operation registers  
896       section, and points to the first Extended Capability structure in a linked list of structures. Each Extended  
897       Capability structure starts with a header containing two fields: a Capability ID, and a Length. The next  
898       Extended Capability structure starts after the end of the previous structure.

899     Extended Capabilities are evaluated by the Driver in the order in which they appear in the linked list.

900     For more details, see *Section 7.3.2*.

## 5.6 Target Device Support Model

### 5.6.1 I3C Devices

901     Per the I3C TCRI Specification at *Section 5.6.1 [MIPI06]*, the Host Controller implementation is required  
902       to support all I3C compliant Devices. However, the Host Controller is not required to support all I3C Bus  
903       speeds or all I3C Modes. A Host Controller implementation that does not support particular I3C Bus speeds  
904       or optional I3C Modes will indicate this via register **HC\_CAPABILITIES** (i.e., using fields **HDR\_TS\_EN** or  
905       **HDR\_DDR\_EN**, see *Section 7.4.4*).

### 5.6.2 I<sup>2</sup>C Devices

906     Per the I3C Specification [*MIPI02*], many types of I<sup>2</sup>C Target Devices can coexist on an I3C Bus, but I<sup>2</sup>C  
907       Bus Controllers cannot coexist on an I3C Bus.

908     **Interrupts:** Legacy I<sup>2</sup>C Devices use out-of-band mechanisms for interrupts, and as a result will neither  
909       generate nor populate IBIs.

910     **Read/Write:** Read/writes for I<sup>2</sup>C Devices are performed in a similar manner as for I3C Devices (see the I3C  
911       TCRI Specification at *Section 5.6.2 [MIPI06]*).

912     **Presence:** If any I<sup>2</sup>C Devices are present on the Bus, then register **HC\_CONTROL** (field **I2C\_DEV\_PRESENT**,  
913       see *Section 7.4.2*) must properly indicate the Mode that is compliant to I<sup>2</sup>C Devices.

This page intentionally left blank.

## 6 Theory of Operation

The Host Controller life cycle is described in the following operations / flows:

- Initialization, Host Controller setup
- Initial bus enumeration phase
- Request handling
- Error handling
- Event handling
- Device Hot-Join handling, enumeration after Hot-Join
- Power management suspend / resume
- Host Controller de-initialization

While some of these flows can be executed in parallel, they are treated separately here for purposes of describing Host Controller operation. Any necessary synchronization between the flows is called out explicitly.

**Note:**

*Unless otherwise noted, all normative and informative subsections within this section assume that the Host Controller is operating in Active Controller mode. Section 6.17 defines the behavior in Standby Controller mode (if supported), as well as the transitions to and from this mode.*

## 6.1 Host Controller Management

### 6.1.1 Host Controller Initialization

**Precondition:** The following steps assume that the Host Controller is coming out of a hardware-initiated or software-initiated reset. This reset may be preceded by a power-up.

The Driver should perform the following steps to initialize and start the Host Controller:

1. Evaluate Host Controller Register Map version, by examining register **HCI\_VERSION** (*Section 7.4.1*).
2. Evaluate base offsets for the DAT and the DCT:

**Note:**

*Optionally, the Host Controller may require the DAT and/or the DCT to be implemented in Driver-allocated memory, if supported by the System Bus. This is indicated with a value of 0 in the offset field. In this case the DAT and DCT entries may optionally be accessed via other request types, and in this case might not be exposed as part of the register map. The Host Controller shall be notified on DAT entry modification with Internal Control command.*

- A. Read register **DAT\_SECTION\_OFFSET** (*Section 7.4.11*).

If field **TABLE\_OFFSET** contains the value 0, then the DAT is not implemented in registers and the Driver shall allocate Device Context memory for it.

- B. Read register **DCT\_SECTION\_OFFSET** (*Section 7.4.12*).

If field **TABLE\_OFFSET** contains the value 0, then the DCT is not implemented in registers and the Driver shall allocate Device Context memory for it.

3. Evaluate base offsets for **PIO\_SECTION\_OFFSET** and/or the Ring Headers Descriptor Section Offset (see *Section 7.6*), to determine whether either or both indicate that such registers and capabilities are implemented:

- A. Read register **RING\_HEADERS\_SECTION\_OFFSET** (*Section 7.4.13*).

If field **SECTION\_OFFSET** contains a value of 0, then the Ring Headers Section and the Ring-related DMA capabilities defined in this Specification are not implemented by the hardware (i.e., no such capability).

- B. Read register **PIO\_SECTION\_OFFSET** (*Section 7.4.14*)

If field **SECTION\_OFFSET** contains a value of 0, then the PIO Section is not implemented (i.e., no such capability), thus the PIO access methods defined by this Specification are not available in the hardware.

4. Evaluate Host Controller capabilities, by reading register **HC\_CAPABILITIES** (*Section 7.4.4*)

5. Optionally, evaluate list of Extended Capability structures, if present (*Section 7.3.2*):

- A. Read register **EXT\_CAPS\_SECTION\_OFFSET** (*Section 7.4.15*)

- B. Evaluate the linked list of Extended Capability structures, until the end of the linked list (i.e., an instance of register **EXTCAP\_HEADER** with field **CAP\_ID** having the value 0x00, per *Section 7.7.1*).

- C. The size of each Extended Capability structure (i.e., the offset in DWORDs to the next Extended Capability structure) is derived from that structure's register **EXTCAP\_HEADER** (i.e., field **CAP\_LENGTH**, per *Section 7.7.1*).

6. In DMA Mode, determine the number of Ring Headers supported:

- A. Read field **MAX\_HEADER\_COUNT\_CAPABILITY** in register **RHS\_CONTROL** (*Section 7.6.1*).

- 968 7. In DMA Mode, for all available Ring Bundles, allocate memory for Rings (Command, Response, IBI  
969 Status, and IBI Data):
  - 970 A. Evaluate the sizes of Transfer Descriptor, Response Descriptor, and IBI Status Descriptor  
971 structures as provided in register **CR\_SETUP**'s fields **XFER\_STRUCT\_SIZE** and **RESP\_STRUCT\_SIZE**  
972 (*Section 7.6.10.1*), and in register **IBI\_SETUP**'s field **IBI\_STATUS\_STRUCT\_SIZE** (*Section 7.6.10.2*).
    - 973 B. Allocate Driver memory according to these structures' sizes, and program the Base Address  
974 registers (i.e., fields **BASE\_LO** and **BASE\_HI**, for registers defined in *Section 7.6.10.12* through  
975 *Section 7.6.10.19*) for all enabled Rings in each specific Ring Header.
- 976 8. In DMA Mode, set up the number of enabled Ring Bundles:
  - 977 A. In register **RHS\_CONTROL**, set field **MAX\_HEADER\_COUNT** (*Section 7.6.1*) to a value lower than, or  
978 equal to, the value of field **MAX\_HEADER\_COUNT\_CAPABILITY**.
- 979 9. In PIO Mode, set up the thresholds for the queues:
  - 980 A. Determine the queue sizes by reading values from registers **QUEUE\_SIZE** (*Section 7.5.7*) and  
981 **ALT\_QUEUE\_SIZE** (*Section 7.5.8*).
    - 982 B. Set the queue thresholds by programming values in register **QUEUE\_THLD\_CTRL** (*Section 7.5.5*)  
983 and register **DATA\_BUFFER\_THLD\_CTRL** (*Section 7.5.6*).
- 984 10. In DMA Mode, set up Ring Sizes for each Ring:
  - 985 A. In register **CR\_SETUP**, program the **RING\_SIZE** field (*Section 7.6.10.1*).
    - 986 B. In register **IBI\_SETUP**, set fields **IBI\_STATUS\_RING\_SIZE**, **CHUNK\_SIZE**, and **CHUNK\_COUNT**  
987 (*Section 7.6.10.2*).
      - 988 C. Allocate memory regions for the Rings, and then update all Ring base pointers (see *Section 6.6.1*  
989 for more details regarding Ring Header configuration.)
  - 990 11. Enable the Host Controller:
    - 991 A. If both PIO Mode and DMA Mode are supported, then write the appropriate value to field  
992 **MODE\_SELECTOR** in register **HC\_CONTROL** (*Section 7.4.2*) to set the desired operating mode.
      - 993 B. In register **HC\_CONTROL**, set bit field **BUS\_ENABLE**.
  - 994 12. Enable Host Controller interrupts:
    - 995 A. In registers **INTR\_STATUS\_ENABLE** (*Section 7.4.8*) and **INTR\_SIGNAL\_ENABLE** (*Section 7.4.9*), set  
996 the mask of enabled interrupts.
  - 997 13. In PIO Mode, enable and start the PIO Queues:
    - 998 A. Enable PIO interrupts by setting the mask of enabled interrupts in registers  
999 **PIO\_INTR\_STATUS\_ENABLE** (*Section 7.5.10*) and **PIO\_INTR\_SIGNAL\_ENABLE** (*Section 7.5.11*).
      - 1000 B. Enable the PIO Queues (if not already enabled) by setting field **ENABLE** in register **PIO\_CONTROL**  
1001 (*Section 7.5.13*).
        - 1002 C. Start the PIO Queues by setting field **RS** in register **PIO\_CONTROL**.
    - 1003 14. In DMA Mode, enable and start the Ring Headers. For each enabled Ring Header:
      - 1004 A. In register **RH\_CONTROL** (*Section 7.6.10.9*), enable the Ring Header by setting the corresponding  
1005 **ENABLE** bit. The Driver shall not modify the Ring Header settings (i.e., base pointers, Ring/chunk  
1006 sizes etc.) upon enabling the Ring Bundle.
        - 1007 B. In registers **RH\_INTR\_STATUS\_ENABLE** (*Section 7.6.10.5*) and **RH\_INTR\_SIGNAL\_ENABLE**  
1008 (*Section 7.6.10.6*), enable interrupts.
          - 1009 C. In register **RH\_CONTROL** (*Section 7.6.10.9*), start the Ring Header by setting bit field **RS**.

1010 The Host Controller shall expose its Register Map and wait for the Driver to program the **BUS\_ENABLE** bit  
1011 field in register **HC\_CONTROL**. Once set, the Host Controller shall evaluate capabilities limited by the  
1012 Driver and initialize internal queues/resources accordingly.

1013 An example of a software-driven limitation is the number of Ring Headers used by the software. The  
1014 Driver shall expose the number of Ring Headers intended to be used, by specifying that value in the  
1015 **MAX\_HEADER\_COUNT** field of register **RHS\_CONTROL**. The Host Controller may power-gate specific  
1016 functionality until the next initialization.

### 6.1.2 Host Controller De-Initialization

The Driver would perform the following steps to tear down the Host Controller:

1. Disable the Controller bus operation:

A. In register **HC\_CONTROL** (*Section 7.4.2*), clear the **BUS\_ENABLE** bit field.

2. In DMA Mode, stop the Controller Rings.

For all running Ring Bundles:

A. In register **RH\_CONTROL** (*Section 7.6.10.9*), clear the **RS** bit field.

3. In DMA Mode, disable the Controller Rings.

For all enabled Ring Bundles:

A. In register **RH\_CONTROL** (*Section 7.6.10.9*), clear the **ENABLE** bit field.

4. For DMA Mode, de-allocate memory allocated for Rings, for all Ring Bundles.

5. For PIO Mode, stop operation of the PIO Queues using register **PIO\_CONTROL** (*Section 7.5.13*):

A. Clear bit field **RS**.

B. Clear bit field **ENABLE**.

### 6.1.3 Host Controller Reset

Host Controller reset is performed with register **RESET\_CONTROL**. In DMA Mode, individual Ring Bundles are reset independently via register **RH\_CONTROL**.

The Driver would perform the following steps to put the Host Controller into reset state:

1. Disable the Controller bus operation:

A. In register **HC\_CONTROL** (*Section 7.4.2*), clear the **BUS\_ENABLE** bit field.

2. In DMA Mode, Read number of enabled Ring Bundles

A. In register **RHS\_CONTROL**, read field **MAX\_HEADER\_COUNT** (*Section 7.6.1*)

3. In DMA Mode, stop the Controller Rings.

For all running Ring Bundles:

A. In register **RH\_CONTROL** (*Section 7.6.10.9*), clear the **RS** bit field.

B. Wait for interrupt

4. In DMA Mode, disable the Controller Rings.

For all enabled Ring Bundles:

A. In register **RH\_CONTROL** (*Section 7.6.10.9*), clear the **ENABLE** bit field.

B. Wait for interrupt

5. In PIO Mode, stop operation of the PIO Queues using register **PIO\_CONTROL** (*Section 7.5.13*):

A. Clear bit field **RS**.

B. Clear bit field **ENABLE**.

6. Reset HC

A. In register **RESET\_CONTROL** (*Section 7.4.5*), set the **SOFT\_RST** bit field.

B. Poll until **SOFT\_RST** bit field is cleared or until an **HC\_INTERNAL\_ERR\_STAT** interrupt is flagged.

#### 6.1.4 Power Management

Power Management is considered to be system implementation specific, and as a result this Specification cannot provide a comprehensive statement on the Host Controller's support for Power Management, for every system or platform implementation. In particular the System Bus, OS, and platform-specific implementations may drive different needs, or pose different limitations, for Host Controller power management flows. For flexibility and effectiveness, Power Management support at the HCI level should be designed to avoid limiting the Host Controller functionality during normal operation.

For most system implementations, a Host Controller must still be responsive to the system's power management approach, especially if the system uses a power management schema that attempts to minimize power consumption upon detection of idleness for attached Devices (including the Host Controller). A lack of activity on the interface that connects the Host Controller to the system (i.e., PCI\*, USB) may not be a reliable indicator that the Host Controller can safely reduce its power allocation without impact to its functionality. For example, the Host Controller may require specific minimal power state to continue processing queued command transactions, or in order to process In-Band Interrupts from Target Devices. As such, the Host Controller may need to carefully manage its link state, using an interface-specific method, or send a remote-wakeup signal to the system during situations when it knows it has received an unexpected Bus event (i.e., an Interrupt from a Target), in order to ensure required power state. As link states and power states are typically closely associated, the Host Controller's interface logic needs to be aware of the current state of any queued command transactions, as well as the arrival of unexpected Bus events, and drive changes appropriately.

In some cases, the Host Controller may be forced into a lower-power state by the system, from which it cannot reliably pull itself back into a higher-power state in order to remain responsive to Bus events. In these cases, the Host Controller shall automatically NACK any Hot-Join Requests, In-Band Interrupt Requests, or Controller Role Requests from any other Devices, even if they would otherwise be permitted and acknowledged.

The Driver expectation is that any Host Controller initiated power management capability is autonomous (i.e., doesn't need Driver intervention). The Driver may choose to manually manage power by disabling / resetting the Controller, in which case Host Controller re-initialization would be taken care of by the Driver. The Driver shall also work within the OS power management framework to keep the Host Controller sufficiently powered when possible, so that it can continue processing queued command transactions, or respond to unexpected Bus events appropriately.

#### 6.1.4.1 Power Management in Standby Controller Mode

A Host Controller that supports Secondary Controller functionality (see [Section 6.17](#)) requires additional support for power management, in order to remain responsive to I3C Bus activity that might signal a potential transfer of the Controller Role. The process for passing the Controller Role, with steps necessary to fully prepare for Controller Role Handoff, is defined in the I3C Specification [[MIPI02](#)] at [Section 5.1.7.1](#). A successful Controller Role Handoff would require the Secondary Controller Logic to remain active and respond to CCCs sent by the Active Controller, so that the Active Controller can determine whether the Secondary Controller Logic needs to be re-synchronized (i.e., after returning from a Deep Sleep state).

A Host Controller that supports Standby Controller mode shall attempt to monitor the Bus for events that might indicate that the Active Controller is trying to pass the Controller Role to it. If field **AC\_FSM\_OP\_SELECT** in register **STBY\_CR\_CONTROL** is set, or if the Host Controller has sent a Controller Role Request, then the Secondary Controller Logic shall remain active and shall respond to CCCs sent by the Active Controller, which may indicate that the Active Controller has started the steps to prepare for handoff. The Secondary Controller Logic shall also monitor the Bus for any Broadcast **DEFTGTS** or **DEFGRPA** CCCs that the Active Controller might send; if these are received, then they would be captured and stored in the IBI Data Queue (for PIO Mode) or an IBI Ring Pair (for DMA Mode) per [Section 6.17.3.2](#).

If field **AC\_FSM\_OP\_SELECT** in register **STBY\_CR\_CONTROL** is not set, then the Secondary Controller Logic shall respond to the **GETSTATUS** Format 1 CCC (i.e., no Defining Byte) with Bits[7:6] set to 2'b11 to indicate that the Host Controller and its Secondary Controller Logic may not participate in the steps to prepare for handoff, and will not accept the Controller Role at this time since it has not been allowed to do so by its connected Host (see the I3C Specification [[MIPI02](#)] at [Section 5.1.9.3.15](#)).

If such a Host Controller supports Power Management capabilities, then it shall also support the **GETSTATUS** Format 2 CCC with the **PRECR** Defining Byte (see the I3C Specification [[MIPI02](#)] at [Section 5.1.7.3](#)) which serves as the definitive indicator that the Active Controller is reading its status, to determine whether it has returned from a Deep Sleep state and needs to be re-synchronized. From the time the Secondary Controller Logic receives and ACKs this CCC, it should remain active and responsive, and not re-enter a Deep Sleep state that would interrupt the re-synchronization and successful preparation for handoff. To prevent any interruption of the re-synchronization, the Host Controller shall decline any attempts by the system to put it into a lower-power state based on assumption of inactivity, when possible. The Driver must also work within the OS power management framework to keep the system sufficiently awake and powered, and avoid going into a sleep state if possible, since the Driver is also a key actor in the process of re-synchronization (as shown by [Figure 33](#)).

The transition to Active Controller mode can be interrupted if the system sends a power management event or link state transition that cannot be declined and that would put the Host Controller or its connected Host into a lower-power state from which it would be unable to complete re-synchronization or commit to actively managing the I3C Bus as the Active Controller. In such cases, any pre-Handoff re-synchronization flows would be interrupted, and the Secondary Controller Logic: shall decline the **GETACCCR** CCC from the Active Controller; shall NACK any subsequent CCCs relating to Handoff preparation; and shall make itself unavailable to receive the Controller Role while in this lower-power state. The Secondary Controller Logic shall also respond to the **GETSTATUS** Format 1 CCC (i.e., no Defining Byte) with Bits[7:6] set to 2'b11 (see the I3C Specification [[MIPI02](#)] at [Section 5.1.9.3.15](#)) to indicate that the Host Controller and its Secondary Controller Logic may not participate in the steps to prepare for handoff, and will not accept the Controller Role at this time due to its lower-power state as well as the power state of its connected system.

## 6.2 Host System Memory Access

If the Host Controller implements support for DMA to access the Host's system memory, then it shall expose pairs of 32-bit registers for the Driver to provide the base address to regions of Driver-allocated memory for specific purposes, and the Host Controller's Memory Access Engine (i.e., DMA engine) shall use these registers to access the regions of memory. Such registers are arranged into **LO** / **HI** pairs, and each register pair shall be concatenated to hold a full 64-bit address pointer for the Host system. The Host Controller shall support up to 64-bit addressing for the Host system's memory address space.

- Register pair **DEV\_CTX\_BASE\_LO** / **DEV\_CTX\_BASE\_HI** (see *Section 7.4.19* and *Section 7.4.20*) together hold the base address pointer for the Device Context, which are used for the DAT and/or the DCT.
- If DMA Mode is supported, then each Ring Header provides four register pairs for the Rings within the associated Ring Bundle:
  - Register pair **RH\_CMD\_RING\_BASE\_LO** / **RH\_CMD\_RING\_BASE\_HI** (see *Section 7.6.10.12* / *Section 7.6.10.13*) together hold the base address pointer for the Command Ring.
  - Register pair **RH\_RESP\_RING\_BASE\_LO** / **RH\_RESP\_RING\_BASE\_HI** (see *Section 7.6.10.14* / *Section 7.6.10.15*) together hold the base address pointer for the Response Ring.
  - Register pair **RH\_IBI\_STATUS\_RING\_BASE\_LO** / **RH\_IBI\_STATUS\_RING\_BASE\_HI** (see *Section 7.6.10.16* / *Section 7.6.10.17*) together hold the base address pointer for the IBI Status Ring.
  - Register pair **RH\_IBI\_DATA\_RING\_BASE\_LO** / **RH\_IBI\_DATA\_RING\_BASE\_HI** (see *Section 7.6.10.18* / *Section 7.6.10.19*) together hold the base address pointer for the IBI Data Ring.
- If Scheduled Commands with Handler Type 3 is supported (per *Section 6.16.4*), then each such instance of the Scheduled Commands logic provides one register pair for the Schedule Buffer:
  - Register pair **SCHEDULE\_BUF\_BASE\_LO** / **SCHEDULE\_BUF\_BASE\_HI** (*Section 7.7.10.7* / *Section 7.7.10.8*) together hold the base address pointer for the Schedule Buffer.

### 6.2.1 Scatter-Gather for Host System Memory

1147 **Note:**

1148     *This section applies only for Host Controller implementations that support access to the Host's*  
1149     *system memory via DMA, and also support additional Scatter-Gather capabilities, for specific*  
1150     *regions of allocated memory. Note that some of these Scatter-Gather capabilities are specific to*  
1151     *DMA Mode.*

1152     In order to support systems with fragmented physical memory, the Host Controller may support additional  
1153     Scatter-Gather capabilities. The Host Controller shall expose the level of support for Scatter-Gather when  
1154     accessing Host system memory for specific purposes, including Rings (for DMA Mode) and Device  
1155     Context. The Host Controller advertises its capabilities with three capability fields in register  
1156     **HC\_CAPABILITIES** (see [Section 7.4.4](#)):

- 1157     • Field **SG\_CAPABILITY\_DC\_EN**: Allows Device Context memory to be physically fragmented
- 1158     • If DMA Mode is supported, then each Ring Header may optionally provide up to four registers for the  
1159         Rings within the associated Ring Bundle:
  - 1160         • Field **SG\_CAPABILITY\_IBI\_EN**: Allows memory for IBI Status and IBI Data Rings to be physically  
1161             fragmented
  - 1162         • Field **SG\_CAPABILITY\_CR\_EN**: Allows memory for Command and Response Rings to be physically  
1163             fragmented

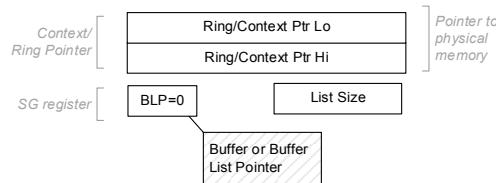
1164     **Note:**

1165         *The Scatter-Gather capability fields for Rings apply to all Ring Bundles in the Host Controller.*  
1166         *As such, the Scatter-Gather registers for Ring Headers shall either be supported for all Ring*  
1167         *Headers, or not supported for any Ring Headers. However, each Scatter-Gather register may*  
1168         *be configured individually for the Ring in conjunction with its LO / HI register pair for such a*  
1169         *Ring's memory region (per [Table 3](#)), if a Ring Header provides the Scatter-Gather register*  
1170         *and if its Ring Bundle supports the Scatter-Gather capability.*

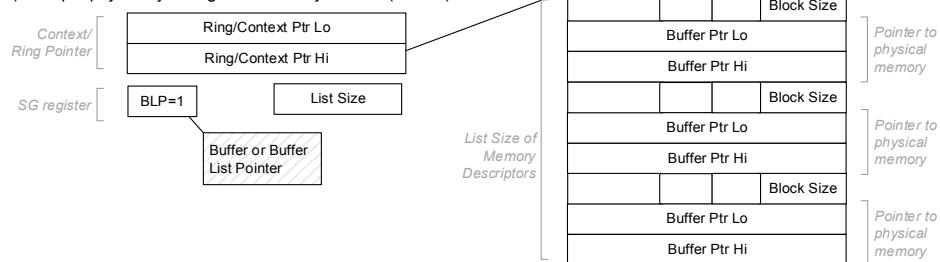
1171     Upon enabling Scatter-Gather for any Ring (or any other context) that supports it, the pointer to physical  
1172     memory shall point to a physically contiguous list of Memory Descriptor structures, with each entry  
1173     describing one chunk of physically contiguous memory. The use of a Scatter-Gather List is indicated in the  
1174     related setup register, in the **BLP** bit. The length of the table, in entries, is provided in the **LIST\_SIZE** field.

1175  
1176 **Figure 6** shows a comparison of the configuration for a single memory chunk vs. for multiple memory chunks.

a) single, physically contiguous memory



b) multiple physically contiguous memory chunks (SG list)



**Figure 6 Scatter-Gather for Ring Context Pointers**

1177

1178 **Table 3** lists registers used to program Scatter-Gather List for supported Rings and other contexts.

1179

**Table 3 Scatter-Gather Support for Rings and Other Contexts**

Purpose	Capability Bit	Base Memory Pointers and Scatter-Gather Setup Register	List Entry Structure
Device Context	<b>SG_CAPABILITY_DC_EN</b>	<b>DEV_CTX_BASE_LO / DEV_CTX_BASE_HI</b> <i>(Section 7.4.19 / 7.4.20)</i> and <b>DEV_CTX_SG</b> ( <i>Section 7.4.21</i> )	Memory Descriptor
Command Ring	<b>SG_CAPABILITY_CR_EN</b>	<b>RH_CMD_RING_BASE_LO / RH_CMD_RING_BASE_HI</b> <i>(Section 7.6.10.12 / 7.6.10.13)</i> and <b>RH_CMD_RING_SG</b> ( <i>Section 7.6.10.20</i> )	Memory Descriptor
Response Ring	<b>SG_CAPABILITY_CR_EN</b>	<b>RH_RESP_RING_BASE_LO / RH_RESP_RING_BASE_HI</b> <i>(Section 7.6.10.14 / 7.6.10.15)</i> <b>RH_RESP_RING_SG</b> ( <i>Section 7.6.10.21</i> )	Memory Descriptor
IBI Status Ring	<b>SG_CAPABILITY_IBI_EN</b>	<b>RH_IBI_STATUS_RING_BASE_LO / RH_IBI_STATUS_RING_BASE_HI</b> <i>(Section 7.6.10.16 / 7.6.10.17)</i> <b>RH_IBI_STATUS_RING_SG</b> ( <i>Section 7.6.10.22</i> )	Memory Descriptor
IBI Data Ring	<b>SG_CAPABILITY_IBI_EN</b>	<b>RH_IBI_DATA_RING_BASE_LO / RH_IBI_DATA_RING_BASE_HI</b> <i>(Section 7.6.10.18 / 7.6.10.19)</i> <b>RH_IBI_DATA_RING_SG</b> ( <i>Section 7.6.10.23</i> )	Memory Descriptor
Schedule Buffer	(None) <i>(Implicitly supported with Handler Type 3)</i>	<b>SCHEDULE_BUF_BASE_LO / SCHEDULE_BUF_BASE_HI</b> <i>(Section 7.7.10.7 / 7.7.10.8)</i> <b>SCHEDULE_BUF_DESCR</b> ( <i>Section 7.7.10.6</i> )	Memory Descriptor

1180

**Note:**

1181  
1182  
1183  
1184  
1185  
1186  
1187  
1188

Scatter-Gather for the Command/Response Rings and IBI Status/Data Rings are optional capabilities. If a Host Controller supports DMA Mode, then it implicitly supports standard Scatter-Gather capabilities for typical Transfer Commands, as indicated by fields in the Data Buffer Descriptor (see **Section 8.3.1**) that allow it to indicate a Buffer List Pointer with non-contiguous memory. These standard Scatter-Gather capabilities were previously defined in earlier versions of this I3C HCI Specification, as a required capability of DMA Mode. The additional capability fields in register **HC\_CAPABILITIES** offer additional Scatter/Gather capabilities that go beyond the standard capabilities.

1189  
1190  
1191  
1192  
1193  
1194  
1195

Similarly, if a Host Controller supports one or more instances of Scheduled Commands logic with Handler Type 3, then it implicitly supports standard Scatter-Gather capabilities for each Schedule Buffer that is used to hold Transfer Commands for this purpose.

For both cases listed above, implementers should ensure that the Memory Access Engine (i.e., DMA engine) can support Scatter-Gather for additional memory regions (such as Device Context, Scheduled Commands, or Rings) when considering whether to also support the optional Scatter-Gather capabilities.

## 6.3 Device Management

The Host Controller provides capabilities for managing the I3C and I<sup>2</sup>C Devices on the I3C Bus, as well as configuring automatic response to various Bus conditions such as Interrupt Requests.

**Note:**

*For additional context, refer to the I3C TCRI Specification at Section 6.1 [[MIPI06](#)].*

### 6.3.1 Device Attach, Enumeration, and Initialization

After the Host Controller is initialized, the initial I3C Bus enumeration should be performed. The Host Controller shall support the Dynamic Address Assignment process, during which the Host Controller assigns Addresses to multiple Devices on the Bus, based on an ENTDAA command issued by the Driver through a Command Descriptor structure. For every Device on the Bus that participates in Dynamic Address Assignment, a single DAT entry must be assigned. Within each DAT entry, field **STATIC\_ADDRESS** is programmed by the Driver, based on *a priori* knowledge of the connected Devices on the Bus (i.e., only those that have an assigned Static Address). Additionally, for every Dynamic Addressing capable Device, field **DYNAMIC\_ADDRESS** shall be set for each valid DAT entry, as part of the Dynamic Address Assignment process.

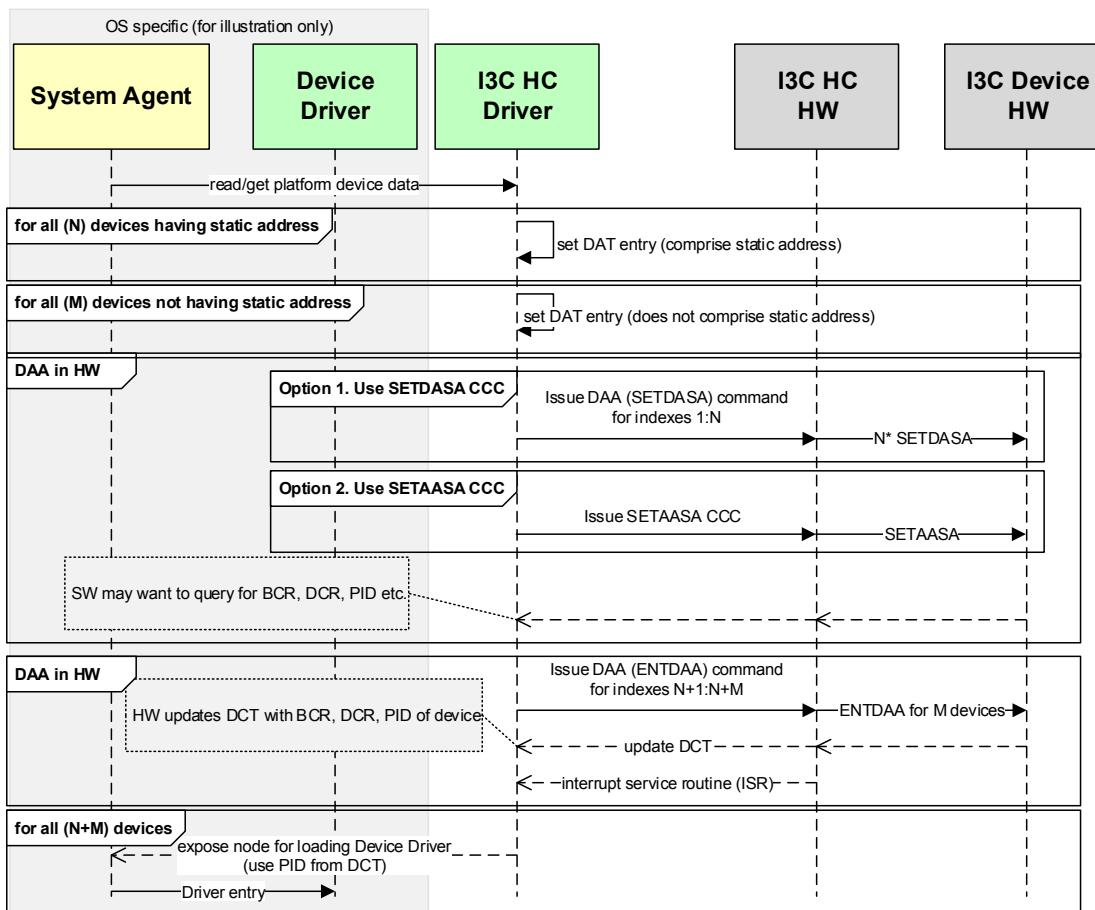
Once the DAT table is set up, the software may choose to assign the Address using either the **SETDASA** CCC (Direct), the **SETAASA** CCC (Broadcast), and/or the **ENTDAA** procedure.

- **SETDASA and SETAASA:** These CCCs rely on prior knowledge of Devices that have Static Addresses assigned.
- **ENTDAA:** When the Host Controller executes the **ENTDAA** command, the first Device that wins Arbitration is assigned the Dynamic Address contained in the first index, as indicated in the Address Assignment Command (field **DEV\_INDEX**). The order of Address assignment is determined by the contents of the DAT table, the provided index in the Address Assignment Command, and Arbitration priority on the Bus. Note that the Driver may choose to address just a subset of the Devices present on the Bus.

The DCT table is updated by Host Controller only when the **ENTDAA** CCC method is used. For the **SETDASA** and **SETAASA** method, the software may read the BCR, the DCR, and the PID with separate transfers to the Device.

1222

**Figure 7** illustrates an example Device enumeration process.



1223

**Figure 7 Example Flow for Initial Bus Enumeration**

For the initial Bus enumeration, the Driver shall perform the following steps:

1. Check the Device Address Table (DAT) and Device Context Table (DCT) sizes.
  - A. Check the size of the DAT table, by reading the **TABLE\_SIZE** and **ENTRY\_SIZE** fields of register **DAT\_SECTION\_OFFSET** ([Section 7.4.11](#)). The **TABLE\_SIZE** field value bounds the overall number of addressable Devices.
  - B. Check the size of the DCT table, by reading the **TABLE\_SIZE** and **ENTRY\_SIZE** fields of register **DCT\_SECTION\_OFFSET** ([Section 7.4.12](#)). The **TABLE\_SIZE** field value bounds the number of entries addressable with a single DAA command (see I3C's **ENTDAA** CCC).
2. Set up the Device Address Table (DAT) entries (see [Section 8.1](#)) with the preferred Dynamic Addresses (for I3C Devices having Static Address), and enqueue appropriate Command Descriptors to assign Dynamic Addresses based on Static Addresses (see [Section 6.4.2](#)).
3. Set up the Device Address Table (DAT) entries (see [Section 8.1](#)) with the preferred Dynamic Addresses (for I3C Devices not having Static Address), and enqueue appropriate Command Descriptors to assign Dynamic Addresses using the **ENTDAA** procedure (see [Section 6.4.1](#)).
4. If any I3C Devices need to change any assigned Dynamic Addresses:
  - A. Enqueue one or more Command Descriptors of Immediate Data Transfer Command type for the **SETNEWDA** Direct CCC, as a standard CCC (i.e., not an Address Assignment Command).
  - B. Wait for a response, using polling or interrupts (as appropriate for the operating mode). Ensure that field **ERR\_STATUS** of the Response Descriptor indicates a successful result.

After the Dynamic Addressing procedure completes, it's possible that some dedicated Devices might not have been assigned the intended Dynamic Addresses (i.e., the Dynamic Address intended for a Device with a specific PID was instead assigned to another Device). For example, this can occur when some Devices are either missing on the Bus, or else not yet ready to be addressed. In some cases, the software may change the Dynamic Address for another Device that was assigned successfully, by enqueueing a Transfer Command with the **SETNEWDA** CCC. In other cases the software must either reset all Dynamic Addresses using the **RSTDAA** Broadcast CCC; repeat the Dynamic Addressing procedure by enqueueing new Address Assignment Commands; or else use the Target Reset Pattern to reset some or all I3C Target Devices on the I3C Bus.

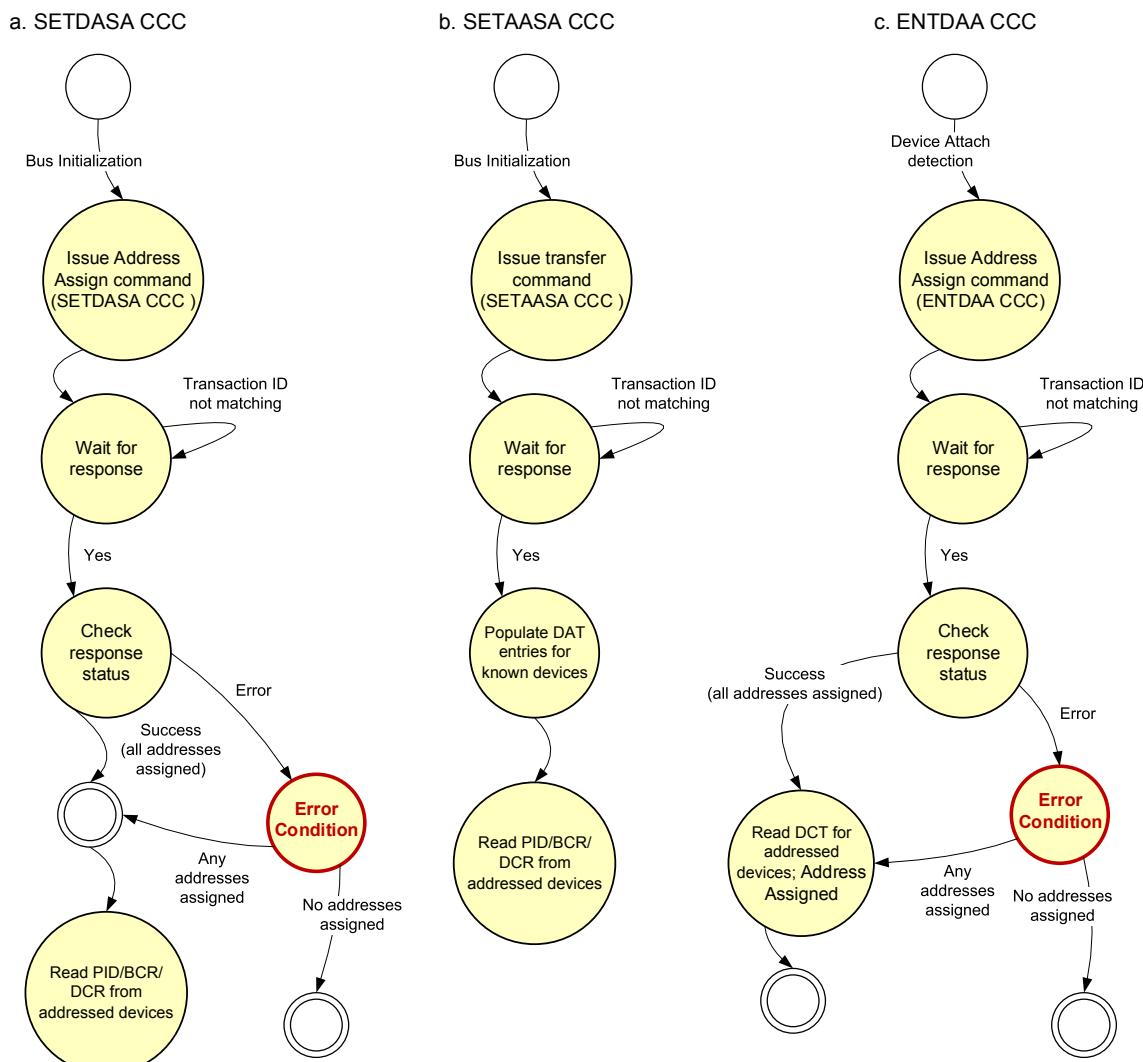


Figure 8 Address Assign Flow

**Note:**

In Figure 8, the **SETAASA** CCC is sent as a Broadcast CCC, using a standard Transfer Command (i.e., not an Address Assignment Command). This limits the possible error status codes that can be returned in the Response Descriptor (e.g., field **ERR\_STATUS** having a value of 0x4) if the Host Controller encounters an error while sending the Broadcast CCC on the I3C Bus.

1258 The Host Controller shall receive and handle Hot-Join Requests from I3C Target Devices. Hot-Join  
1259 Requests shall be automatically handled (i.e., either ACKed or NACKed) according to the current value of  
1260 field **HOT\_JOIN\_CTRL** in register **HC\_CONTROL** (see *Section 7.4.2*). Upon receiving a Hot-Join Request  
1261 from an I3C Target, the Host Controller shall conditionally notify the Host, by generating an IBI Status  
1262 Descriptor (see *Section 8.6*). Such an IBI Status Descriptor shall indicate the Hot-Join ID in field **IBI\_ID**,  
1263 with a value of 7'h02 (i.e., the reserved Hot-Join Address) in Bits[15:9], and 1'b0 (i.e., RnW = 0) in Bit[8].  
1264 If the Hot-Join Request was rejected (i.e., NACKed), then the Host Controller shall conditionally notify the  
1265 Host, according to the current value of field **NOTIFY\_HJ\_REJECTED** in register **IBI\_NOTIFY\_CTRL** (see  
1266 *Section 7.4.17*).

1267 **Note:**

1268 *The I3C Bus Controller Logic does not distinguish whether an I3C Target might have used either  
1269 the standard method or a passive method to send a Hot-Join Request (see version 1.1.1+ of the  
1270 I3C Specification [MIPI05], Section 5.1.5). In effect, any valid IBI from the reserved Hot-Join  
1271 Address of 7'h02 / W shall be handled as a Hot-Join Request, regardless of the method that the  
1272 I3C Target might use.*

1273 In response to an incoming Hot-Join IBI, the software shall populate a single new entry in the DAT table,  
1274 and shall issue an Address Assignment Command pointing to this entry in the DAT table, using fields  
1275 **DEV\_COUNT** and **DEV\_INDEX**. Since Hot-Join requires Dynamic Address Assignment with the ENTDAAA  
1276 procedure, software shall assign an initial Dynamic Address in field **DYNAMIC\_ADDRESS**.

1277 To schedule the transfer, the software indicates the target Device Address by using its index in the DAT  
1278 table. The Host Controller shall use the appropriate Address from the DAT table for the Bus transaction,  
1279 depending upon the Device type:

- **For Transactions to I3C Devices:** The Dynamic Address shall be used, if defined.

1281 Exceptions:

- **Address Assignment Command with SETDASA CCC:** This is always sent to the Static Address.
- **Address Assignment Command with ENTDAA CCC:** The ENTDAA procedure is used, and the initial  
Dynamic Address (i.e., in field **DYNAMIC\_ADDRESS**) is assigned to the Device during this procedure.
- **Transfer Commands that are Broadcast CCCs:** No DAT entry is used, per *Section 6.7*. This includes  
the SETAASA CCC.

- **For Transactions to I<sup>2</sup>C Devices:** The Static Address shall always be used.

1288 The Driver shall not change the DAT table entry's Dynamic Address while the Device is in the Addressed  
1289 state, with exception of a successful completion of an Address change request (i.e., using the SETNEWDA  
1290 CCC).

1291 As field **MODE** is not present for the Address Assignment command, the HCI controller shall automatically  
1292 select the appropriate transmission rate, based on the CCC provided:

- For the SETDASA or SETAASA CCC: Push-Pull Mode
- For the ENTDAA CCC: Open-Drain Mode

1295 Any I<sup>2</sup>C Devices that might be present on the Bus do not take part in dynamic enumeration. To set up a  
1296 DAT entry for I<sup>2</sup>C Devices:

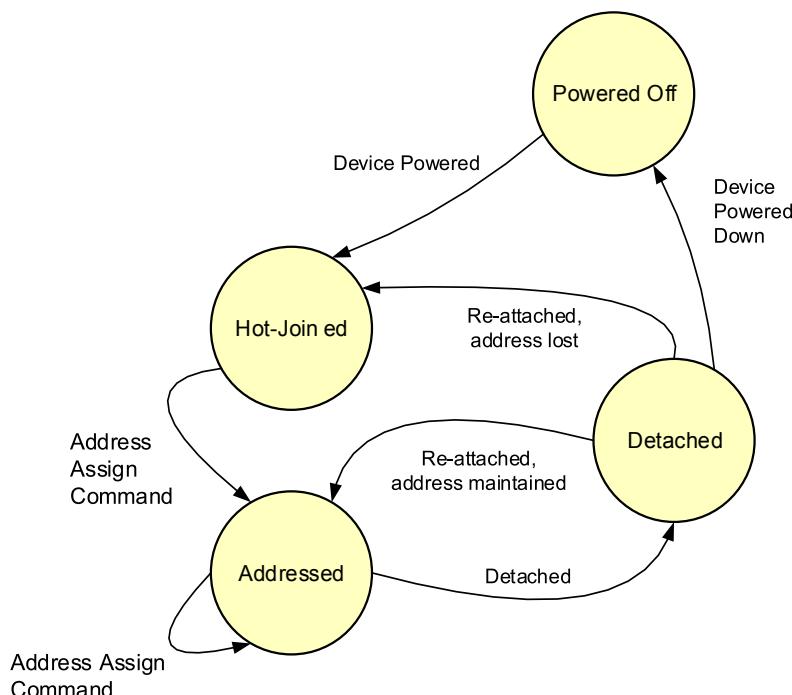
- Set the **DEVICE** field to indicate the Device's type (i.e., I<sup>2</sup>C).
- Set the **STATIC\_ADDRESS** field to indicate the Static Address, which is configured by the Bus designer.

### 6.3.2 Device Detach, Reset, and Power Management

Per the I3C Specification [[MIPI02](#)], the Controller-capable Device is not aware of a Device detaching from the Bus (including powering down the Device).

Upon a Device detach event (i.e., an unexpected removal of a Device from the Bus) the DAT shall not be altered. The assigned Address is reserved (see the I3C Specification at [Section 5.1.4](#)). The Driver can re-use the specific entry (e.g. if it needs to recycle it) by executing an Address Assignment command using the specific index of the DAT table. In this operation, Device discovery (re-enumeration) may be required. If the Device is gracefully removed, then the software shall clear the DAT table entry.

Once an I3C Device joins the Bus or exits from reset, it shall wait for Dynamic Address assignment. The Driver shall assign a Dynamic Address to the Device, and once a Dynamic Address has been assigned to it, the Device shall respond to all transfers addressed to it. The Device may be driven to a power-gated state, or it may enter a detached state. If the Device retains its Address while in the detached state (as in an Offline Capable Device), then it can re-attach using its existing Address. Optionally, if the Device loses its Address then it can go through the regular enumeration flow.



**Figure 9 Device State FSM**

Because Offline Capable Devices might not instantly respond to directed commands, it is essential for the software to allow Devices to take time to respond, with retries and/or longer timeout. Functionally, from the HCI perspective, such Devices are in the Addressed state.

### 6.3.3 Device Context

The context for Devices present on the Bus is realized through the standard Device Context tables, including the DAT (see *Section 8.1*) and the DCT (see *Section 8.2*).

The standard DAT contains information on Devices on the Bus that could be the target of a command or transfer. Due to the nature of the I3C Bus, specific Devices may be described in dedicated DAT entries, even though they might currently be either detached from the Bus, or not responding according to the I3C protocol. DAT entries also contain fields that control the Host Controller's automatic response to certain types of Interrupt Requests.

The DCT is a transient table of entries, having fields intended to be populated by the Host Controller during the Dynamic Address Assignment procedure with the **ENTDAA** CCC. Note that the DCT size could be significantly smaller than the DAT size. Since DCT entries can be reused for new Devices that would need to have a Dynamic Address assigned, the Driver shall copy the contents of a DCT entry to the internal Driver context, on a per-Device basis.

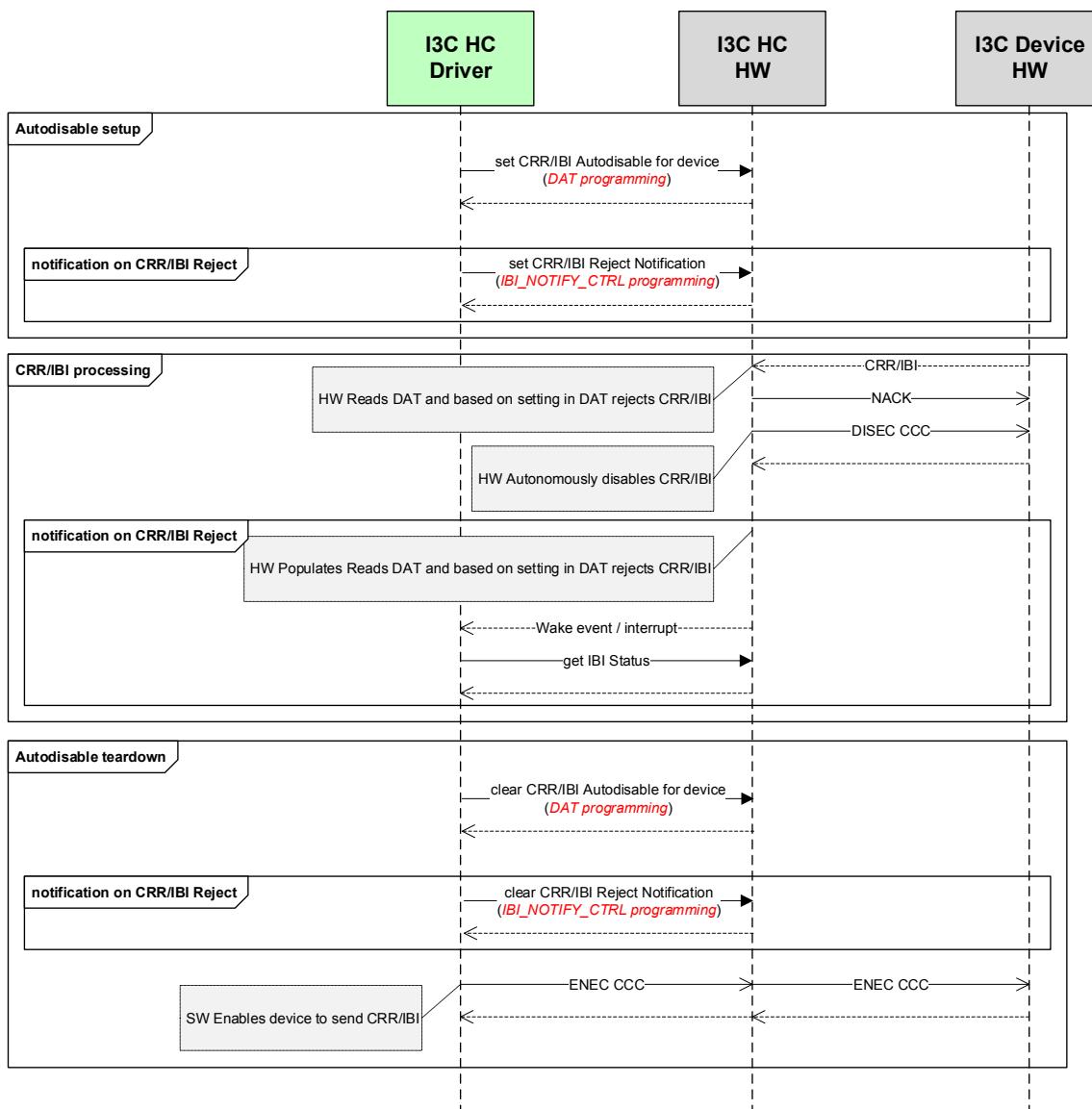
A DAT table entry is associated with a given DCT table entry if the DAT entry's **DYNAMIC\_ADDRESS** field has the same value as the DCT entry's **DYNAMIC\_ADDRESS** field.

The Host Controller supports automatic disabling of Controller Role Request (CRR) or In-Band Interrupt (IBI) from specific Devices, using the **CRR\_REJECT** or **IBI\_REJECT** fields of the DAT table, and also using the Target's credit counter (i.e., if the Target IBI credit counting mechanism is supported and enabled, per *Section 6.9.5*).

- If CRRs are disabled using the **CRR\_REJECT** field of the DAT table, then the software is responsible for re-enabling CRRs using the **ENEC** CCC.
- If IBIs are disabled using the **IBI\_REJECT** field of the DAT table, then the software is responsible for re-enabling IBIs using the **ENEC** CCC.
- If IBIs are disabled using the Target IBI credit counting mechanism (i.e., if supported and enabled) in response to a NACKed IBI Request when the Target's credit counter is zero, then the Host Controller shall automatically re-enable IBIs for that Target using the **ENEC** CCC, once the Host Controller increments the Target's credit counter, i.e., after the Driver writes to that Target's register **TARGET\_CREDIT\_N** (see *Section 7.7.5.3*).

*Figure 10* shows the flow for the Host Controller automatically disabling Controller Role Requests and In-Band Interrupts based on DAT entries.

Optionally, the software may configure an interrupt source for getting information on whether Controller Role Requests and/or In-Band Interrupts might have been disabled (i.e., fields **NOTIFY\_CRR\_REJECTED** and **NOTIFY\_IBI\_REJECTED** in register **IBI\_NOTIFY\_CTRL**, see *Section 7.4.17*). In this case, the IBI Status Queue/Ring shall be populated with notifications of NACKed CRRs and/or IBIs.



1349

**Figure 10 Controller-Role Request (CRR) / In-Band Interrupt (IBI) Autodisable Flow**

## 6.4 Device Addressing

The Host Controller supports the I3C-defined methods of assigning a Dynamic Address to I3C Target Devices:

- The Address Assignment Command type of the Command Descriptor (see [Section 8.4.1](#)) shall be used to perform Dynamic Address Assignment using either the [ENTDAA](#) CCC or the [SETDASA](#) CCC.
- The Immediate Data Transfer Command (see the I3C TCRI Specification [[MIPI06](#)] at [Section 7.1.2.1](#)) shall be used to perform Dynamic Address Assignment using the [SETAASA](#) CCC, which is a Broadcast CCC that applies to all such Target Devices on the Bus.

The Host shall initiate and direct the steps for Dynamic Address Assignment, using software to enqueue the appropriate Command Descriptors. Once an I3C Target Device has been assigned a Dynamic Address, software may also use the [SETNEWDA](#) CCC to change the assigned Dynamic Address, by enqueueing an Immediate Data Transfer Command type of the Command Descriptor.

**Note:**

*If the Transfer Command that sends the [SETNEWDA](#) CCC is successful, then software must then update the appropriate DAT entry for that I3C Target Device to use the new Dynamic Address, before enqueueing any new Transfer Commands intended for that I3C Target Device.*

*In DMA Mode, if the DAT entry has not yet been updated, and if the Host Controller receives any incoming IBIs from such an I3C Target Device that has recently changed its Dynamic Address, then such IBI Requests would be sent to Ring Bundle 0 (per [Section 6.6.1](#) and [Section 6.9.2](#)), as there would not yet be any dedicated DAT entry with the new Dynamic Address.*

The Host Controller may also use the [RSTDAA](#) CCC to reset the Dynamic Address of all I3C Target Devices. After using this Broadcast CCC, all assigned Dynamic Addresses would be reset, and any DAT entries for such I3C Target Devices should be reused or updated, as they are likely to become invalid.

The Host Controller also supports I<sup>2</sup>C Target Devices, which have Static Addresses that are assigned by the Bus Designer (i.e., with *a priori* knowledge) and which cannot be changed with I3C standard CCCs. If such I<sup>2</sup>C Target Devices are known to be on the I3C Bus, then the Host:

- Shall not attempt to use any of the I3C standard CCCs for Dynamic Address Assignment to assign these same Static Addresses to any I3C Target Devices, and
- Shall not attempt to use the [SETNEWDA](#) CCC to change the assigned Dynamic Address of any I3C Target Device to conflict with any of these same Static Addresses.

#### 6.4.1 Dynamic Address Assignment with ENTDAA

For I3C Target Devices that support the ENTDAA procedure and that have not been assigned Static Addresses by the Bus Designer, software shall issue Command Descriptors to assign Dynamic Addresses, according to the defined Bus Initialization Sequence with Dynamic Address Assignment (see **Section 5.1.4.2** of the I3C Specification [[MIP102](#)]).

Each I3C Target Device that will be used with the ENTDAA Procedure shall have its own DAT entry, populated by software.

For each such DAT entry, software shall:

- Set the **DEVICE** field to indicate the Device's type (i.e., I3C).
- **For DMA Mode:** Set the **RING\_ID** field to indicate which Ring Bundle (in DMA Mode only) will be used to receive IBIs from this Device.
- Set the **DYNAMIC\_ADDRESS** field to indicate the Device's preferred Dynamic Address.

1. Enqueue one or more Command Descriptors of Address Assignment Command type for the **ENTDAA** CCC, using the steps listed in **Section 8.4.1.1**.

2. Wait for a response, using polling or interrupts (as appropriate for the operating mode). Ensure that the Response Descriptor indicates a successful result (i.e., field **ERR\_STATUS**).

3. For each successful response: read the **PID**, **BCR** and **DCR** values from the appropriate fields of the indicated entries in the DCT (see **Section 8.2**) for the assigned Dynamic Address(es) as part of the **ENTDAA** process.

A. Note that these values are transient, so software must read them and save them internally before performing subsequent Address Assignment commands with the **ENTDAA** CCC.

B. Each DCT entry for a successful assignment with the **ENTDAA** modal flow shall have the same value in field **DYNAMIC\_ADDRESS** as the corresponding DAT entry that was used for the I3C Device to which this Dynamic Address was assigned.

If the Address Assignment command is successful, then the Host Controller shall generate a Response Descriptor with a **SUCCESS** response in the Response Descriptor's **ERR\_STATUS** field (see **Section 8.4.1.1** and **Section 8.5**).

In specific cases, the Address Assignment Command might complete with a **NACK** response in field **ERR\_STATUS** of the Response Descriptor, for these reasons:

- A **NACK** on the I3C Broadcast Address (7'h7E) with Read during the **ENTDAA** modal flow indicates that no I3C Devices are available for participation in the DAA (**ENTDAA**) procedure.
  - If the ENTDAA procedure did assign some Dynamic Addresses, this will be indicated in the Response Descriptor, with field **DATA\_LENGTH** having a value greater than zero (i.e., the number of remaining entries before the NACK) but less than field **DEV\_COUNT** of the Address Assignment Command.
  - If the ENTDAA procedure did not assign any Dynamic Addresses, this will be indicated in the Response Descriptor, with field **DATA\_LENGTH** equal to field **DEV\_COUNT** of the Address Assignment Command (i.e., no I3C Devices responded).

**Note:**

*If the I3C Bus Controller Logic sees a NACK on the I3C Broadcast Address (7'h7E) before the **ENTDAA** Broadcast CCC (which usually indicates an Address Header error due to no I3C Devices responding to CCC framing in SDR Mode), then it shall indicate this status by setting field **ERR\_STATUS** to the value 0x4 (i.e., **ADDR\_HEADER**, see **Section 6.13.1.1**).*

After assignment, software may use the **SETNEWDA** CCC with an Immediate Data Transfer Command to change the assigned Dynamic Address for any I3C Target Device, per **Section 6.3.1**.

#### 6.4.2 Using Static Addresses

For I3C Target Devices that have been assigned Static Addresses by the Bus Designer, software shall issue Command Descriptors to assign Dynamic Addresses according to the defined Bus Initialization Sequence with Dynamic Address Assignment (see *Section 5.1.4.2* of the I3C Specification [*MIPI02*]).

Each I3C Target Device that has an assigned Static Address shall have its own DAT entry. I3C Target Devices that are assigned with the Address Assignment Command for SETDASA must be populated in advance with the known Static Address.

For each such DAT entry, software shall:

- Set the **DEVICE** field to indicate the Device's type (i.e., I3C).
- **For DMA Mode:** Set the **RING\_ID** field to indicate which Ring Bundle (in DMA Mode only) will be used to receive IBIs from this Device.
- Set the **DYNAMIC\_ADDRESS** field to indicate the Device's preferred Dynamic Address.

1. For those I3C Devices which will be configured with the **SETDASA** CCC:

- A. Use prior configuration from the Host to set up one DAT entry for each known I3C Target Device that will be configured with the **SETDASA** CCC. Each DAT entry shall include the assigned Static Address in field **STATIC\_ADDRESS**.
- B. Enqueue one or more Command Descriptors of Address Assignment Command type for the **SETDASA** CCC, using the steps listed in *Section 8.4.1.2*.
- C. Wait for a response, using polling or interrupts (as appropriate for the operating mode). Ensure that the Response Descriptor indicates a successful result (i.e., field **ERR\_STATUS**).

2. For those I3C Devices which will be configured with the **SETAASA** CCC:

- A. Use prior configuration from the Host to set up one DAT entry for each known I3C Target Device that will be configured with the **SETAASA** CCC.
- B. Enqueue one Command Descriptor of Immediate Data Transfer Command type for the **SETAASA** Broadcast CCC, as a standard CCC (i.e., not an Address Assignment Command).
- C. Wait for a response, using polling or interrupts (as appropriate for the operating mode). Ensure that the Response Descriptor indicates a successful result (i.e., field **ERR\_STATUS**).

If the Address Assignment command for **SETDASA** is successful, then the Host Controller shall generate a Response Descriptor with a **SUCCESS** response in field **ERR\_STATUS** (see *Section 8.4.1.2* and *Section 8.5*).

In specific cases, the Address Assignment Command for **SETDASA** might complete with a **NACK** response in the Response Descriptor's **ERR\_STATUS** field. An explicit **NACK** on the **SETDASA** Direct CCC might indicate that the I3C Device with the indicated Static Address is not present, or was not ready, or did not detect the **SETDASA** CCC on the I3C Bus.

**Note:**

*If the I3C Bus Controller Logic sees a NACK on the I3C Broadcast Address (7'h7E) before the SETAASA Broadcast CCC or the SETDASA Direct CCC (which usually indicates an Address Header error due to no I3C Devices responding to CCC framing in SDR Mode), then it shall indicate this status by setting field **ERR\_STATUS** to the value 0x4 (ADDR\_HEADER, see *Section 6.13.1.1*).*

If the Immediate Data Transfer Command for the **SETAASA** CCC is successful, then the Host Controller shall conditionally generate a Response Descriptor with a **SUCCESS** response in field **ERR\_STATUS**, just as it would for any other Broadcast CCC that would be sent via a Transfer Command. Note that the **SETAASA** CCC has no method for determining the status of any individual I3C Target Device that is asked to assign its own Static Address as a Dynamic Address.

After assignment, software may choose to use the **SETNEWDA** CCC with an Immediate Data Transfer Command to change the assigned Dynamic Address for any I3C Target Device, per *Section 6.3.1*.

#### 6.4.3 Grouped Addressing

In Active Controller mode, software may issue Command Descriptors that are Transfer Commands (see [Section 6.7](#)) to manage the assignment of I3C Devices to any Group Addresses. Software shall track the Group Addresses and maintain the assignments in memory. I3C Group Addresses share the same address space as valid I3C Dynamic Addresses, so software must avoid assigning conflicting addresses (i.e., a Dynamic Address and a Group Address having the same value).

The following CCCs may be used:

- To assign an I3C Device to a Group Address, software may issue a Direct [SETGRPA](#) CCC addressed to a Dynamic Address (see the I3C Specification [[MIPI02](#)] at [Section 5.1.9.3.27](#)).
- To remove I3C Devices from a Group Address, software may issue a Direct [RSTGRPA](#) CCC addressed to either a Dynamic Address or the Group Address (see the I3C Specification [[MIPI02](#)] at [Section 5.1.9.3.28](#)).
- To disband all Groups, software may issue a Broadcast CCC for the [RSTGRPA](#) CCC (see the I3C Specification [[MIPI02](#)] at [Section 5.1.9.3.28](#)).

**Write-Type Transfers:** A Group Address may be used in the same manner as any Target's Dynamic Address for a Write-Type Transfer Command. Each Group Address must have a dedicated DAT entry in order to be used for Transfer Commands. Software shall only enqueue any Write-Type Transfer Commands directed to that Group Address if a dedicated DAT entry already exists for that Group Address.

**Read-Type Transfers:** Software shall not enqueue a Command Descriptor that would attempt to perform a Read-Type transfer directed to a Group Address, because the I3C Specification specifies that Group Addresses shall only be used for Write-Type transfers (see the I3C Specification [[MIPI02](#)] at [Section 5.1.4.4](#)).

## 6.5 PIO Queue Management

1488 **Note:**

1489     *This section applies only for Host Controller implementations that implement PIO Mode, and thus*  
1490     *accept all Transfer Commands and associated data via register reads and writes. PIO Mode does*  
1491     *not require connection to a System Bus that supports DMA.*

1492     In the I3C HCI Register Map, the PIO Section Offset (see *Section 7.4.14*) register provides the starting  
1493     location of the PIO Register Section. The PIO Register section is the structure of registers that allows for  
1494     scheduling transfers and reception of IBI without need of memory access.

1495     The PIO Register section includes four mailbox registers that are used to accept all Transfer Commands  
1496     and associated data:

- 1497         • A write-only register as the Command Queue Port (i.e., register **COMMAND\_QUEUE\_PORT**,  
1498             *Section 7.5.1*) that accepts Command Descriptor structures (*Section 8.4*).
- 1499         • A read-only register as the Response Queue Port (i.e., register **RESPONSE\_QUEUE\_PORT**,  
1500             *Section 7.5.2*) that provides Response Descriptor structures (*Section 8.5*).
- 1501         • A bi-directional port register as the Transfer Data Port (i.e., register **XFER\_DATA\_PORT**, *Section 7.5.3*)  
1502             that accepts write (Tx) data or provides read (Rx) data. The data queues accessed with this Port are called  
1503             the Tx Data Queue and Rx Data Queue.
- 1504         • A read-only register as the IBI Port (i.e., register **IBI\_PORT**, *Section 7.5.4*) that provides IBI Status  
1505             Descriptor (*Section 8.6*) and IBI payload data.

1506     In addition to these PIO port registers, the PIO Register section contains one or more read-only registers  
1507     that provide information on sizes of the PIO queues, i.e., register **QUEUE\_SIZE** (*Section 7.5.7*) and register  
1508     **ALT\_QUEUE\_SIZE** (*Section 7.5.8*), and a set of registers that allow for programming thresholds for efficient  
1509     batch-based operation. The PIO thresholds are described in *Section 6.5.5*.

1510 **Note:**

1511     *In version 1.1 of this I3C HCI Specification, the Command Queue and Response Queue were*  
1512     *required to have the same size. In this version of the I3C HCI Specification, these sizes may be*  
1513     *defined independently; if so, then register **ALT\_QUEUE\_SIZE** shall report the size of the Response*  
1514     *Queue, if it is different from the size of the Command Queue. Implementers may choose whether to*  
1515     *define these sizes independently, or whether to keep them linked together (i.e., in the same manner*  
1516     *as version 1.1 of this I3C HCI Specification).*

1517     *However, in all other aspects, Command Queue and Response Queue operations in PIO Mode are*  
1518     *the same as in previous versions of this I3C HCI Specification.*

1519     The Command Queue and Response Queue may support up to 255 Commands and Responses,  
1520     respectively. The maximum size of the Transfer Data Queue is 256 DWORDs in each direction.

1521     The set of PIO-specific interrupt registers shall be used for enabling and reporting interrupts. The interrupts  
1522     are related to threshold conditions, as well as indications of possible transfer errors. On detecting any  
1523     transfer error, the Host Controller transitions to Suspended state (i.e., field **RESUME** in register  
1524     **HC\_CONTROL** has a read value of 1'b1; see *Section 7.4.2*). The operation of the PIO Queues can be aborted  
1525     by software as needed, by writing a value of 1'b1 to the **ABORT** field in register **PIO\_CONTROL**. The  
1526     operation of the PIO Queues can also be paused or disabled by software, by writing to fields **RS** and/or  
1527     **ENABLE** in register **PIO\_CONTROL** (see *Section 7.5.13*).

1528     There is only one instance of a PIO Register section (i.e., a single set of PIO queues) per Bus Controller  
1529     instance. Registers **HC\_CONTROL** and **PIO\_CONTROL** shall determine the operating status of the PIO queues.  
1530     The PIO queues are running if the following conditions are true:

- 1531         • If both PIO Mode and DMA Mode are supported: Field **MODE\_SELECTOR** in register **HC\_CONTROL** is  
1532             set to **PIO** (see *Section 5.3* for details)
- 1533         • Field **BUS\_ENABLE** in register **HC\_CONTROL** is set to **ENABLED** (not **DISABLED**).

1534     This enables all operation involving the I3C Bus Controller Logic.

- 1535 • Field **RESUME** in register **HC\_CONTROL** is set to **RUNNING** (not **SUSPENDED**).
- 1536 • If the Host Controller was in the **Halt** state due to an error, then writing a value of 1'b1 shall resume operations from the **Halt** state.
- 1537
- 1538 • Field **ENABLE** in register **PIO\_CONTROL** is set to **ENABLED** (not **SUSPENDED**).
- 1539     This enables access to the PIO Queues via the registers in the PIO Register section. Once 1540     enabled, if the I3C Bus Controller receives In-Band Interrupt Requests from any I3C 1541     Targets, then these will be populated into the IBI Queue (per **Section 6.5.4**).
- 1542 • Field **RS** in register **PIO\_CONTROL** is set to 1'b1 (i.e., running)
- 1543     If set to run, this enables processing of any Command Descriptors that are enqueued into 1544     the Command Queue, and allows the Host Controller to populate Response Descriptors 1545     into the Response Queue (per **Section 6.5.2** and **Section 6.5.3**).

**Note:**

1546     In version 1.1 of this I3C HCI Specification, the operation of PIO Queues was always implicitly 1547     enabled if the Host Controller was enabled (i.e., via register **HC\_CONTROL**) and PIO Mode (not 1548     DMA Mode) was selected. This version of the HCI Specification provides more finely-grained 1549     control over PIO Queue operation and Command Descriptor processing, using register 1550     **PIO\_CONTROL**. The structure of register **PIO\_CONTROL** is similar to the equivalent register 1551     **RH\_CONTROL** for a Ring Bundle.

1552     For reference, in version 1.0 of this I3C HCI Specification, the operating mode for a Host Controller 1553     that supported both PIO Mode and DMA Mode was determined by the number of enabled Ring 1554     Headers, such that PIO Mode was only enabled if no Ring Headers were enabled. This version of 1555     the HCI Specification defines a field in register **HC\_CONTROL** to explicitly set the operating mode. 1556     See **Section 5.3** for more details.

1557     Figure 11 shows the FSM for enabling/disabling the PIO Queues and determining whether the 1558     Command/Response Queues are operational.

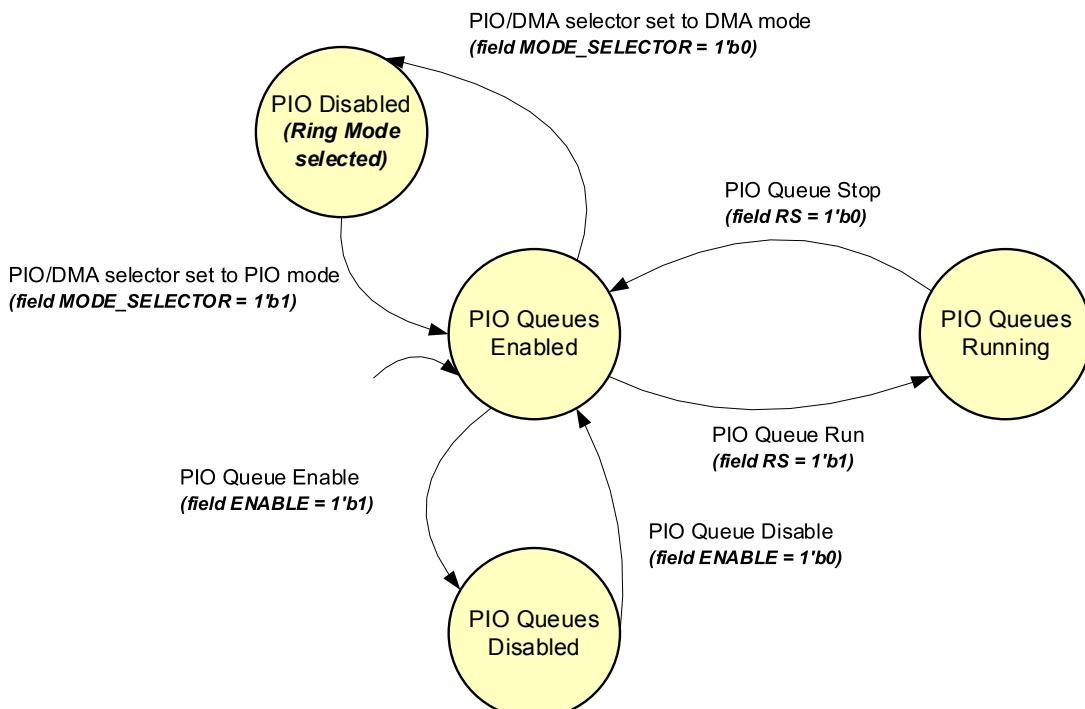


Figure 11 PIO Section FSM

1561 The default state after a reset (if PIO Mode is selected) is:

- 1562 • Field **ENABLE** in register **PIO\_CONTROL** is set to **ENABLED** (1'b1); and  
1563 • Field **RS** in register **PIO\_CONTROL** is set to 1'b1 (i.e., stopped).

1564 The Debug Specific (ID = 0x0C) Extended Capability (if implemented) might provide additional  
1565 information on the state of the PIO queues, and software may read this additional information from the  
1566 Queue Status Level register (i.e., register **QUEUE\_STATUS\_LEVEL**, see *Section 7.7.7.1*) and Data Buffer  
1567 Status Level register (i.e., register **DATA\_BUFFER\_STATUS\_LEVEL**, see *Section 7.7.7.2*). Software may read  
1568 these registers to understand when the PIO Queues have drained to completely empty levels, when batch  
1569 sizes are set to larger values.

1570 In PIO Mode, the IBI data payload is described with **DATA\_LENGTH** of the IBI Status Descriptor structure  
1571 (see *Section 8.6*). This limits number of IBI data to a maximum of 255 bytes, per each IBI Status  
1572 Descriptor. The Host Controller may construct a chain of multiple IBI Status Descriptors, linked together  
1573 with field **LAST\_STATUS**, for a single IBI with a larger payload than 255 bytes.

1574 For each Command Descriptor that is enqueued via the Command Queue Port register, the Host Controller  
1575 shall conditionally generate one Response Descriptor that may be read via the Response Queue Port  
1576 register:

- 1577 • For Write-Type transfers (i.e., when field **RnW** in the Command Descriptor structure has a value of  
1578 1'b0):  
1579 • The Response Descriptor is optional, and shall only be generated when a Transfer Command generates  
1580 an error, or when field **WROC** in the Command Descriptor structure has a value of 1'b1.  
1581 • For Read-Type transfers (i.e., when field **RnW** in the Command Descriptor structure has a value of  
1582 1'b1):  
1583 • The Response Descriptor is required, and shall always be generated.

1584 Software may individually reset specific PIO Queues, using fields in register **RESET\_CONTROL** (see  
1585 *Section 7.4.5*).

### 6.5.1 PIO Section Initialization

Before software may use the PIO Queues to enqueue Transfer Commands, software needs to know the size of each queue, and optionally set batch thresholds for optimal performance.

To initialize a PIO Mode, the Driver shall:

1. Get the sizes of the Command FIFO, Response FIFO, and related Data FIFOs from registers **QUEUE\_SIZE** (*Section 7.5.7*) and (if present:) **ALT\_QUEUE\_SIZE** (*Section 7.5.8*):
  - A. Read **QUEUE\_SIZE** register, evaluate **CR\_QUEUE\_SIZE**.
  - B. Read **QUEUE\_SIZE** register, evaluate **RX\_DATA\_BUFFER\_SIZE**.
  - C. Read **QUEUE\_SIZE** register, evaluate **TX\_DATA\_BUFFER\_SIZE**.
  - D. If register **ALT\_QUEUE\_SIZE** is present, read it and evaluate field **ALT\_RESP\_QUEUE\_SIZE** (i.e., if field **ALT\_RESP\_QUEUE\_EN** is 1'b1).
  - E.
2. If any IBI is enabled, then get the size of IBI FIFO from register **QUEUE\_SIZE** (see *Section 7.5.7*) and (if present:) register **ALT\_QUEUE\_SIZE** (*Section 7.5.8*):
  - A. Read **QUEUE\_SIZE** register, evaluate **IBI\_STATUS\_SIZE**.
  - B. If register **ALT\_QUEUE\_SIZE** is present, then read it and evaluate field **EXT\_IBI\_QUEUE\_EN** (i.e., if field **EXT\_IBI\_QUEUE\_EN** is 1'b1).
3. Optionally, set the thresholds of the Command Queue, Response Queue and related Transfer Data FIFOs:
  - A. Write **QUEUE\_THLD\_CTRL** register, set **CMD\_EMPTY\_BUF\_THLD** field.
  - B. Write **QUEUE\_THLD\_CTRL** register, set **RESP\_BUF\_THLD** field.
  - C. Write **DATA\_BUFFER\_THLD\_CTRL** register, set **TX\_BUF\_THLD** field.
  - D. Write **DATA\_BUFFER\_THLD\_CTRL** register, set **RX\_BUF\_THLD** field.
4. Optionally, set the starting thresholds of the Data FIFOs:
  - A. Write **DATA\_BUFFER\_THLD\_CTRL** register, set **TX\_START\_THLD** field.
  - B. Write **DATA\_BUFFER\_THLD\_CTRL** register, set **RX\_START\_THLD** field.
5. If any IBI is enabled: optionally, set the thresholds of the IBI Queue:
  - A. Write **QUEUE\_THLD\_CTRL** register, set **IBI\_STATUS\_THLD** field.
  - B. Write **QUEUE\_THLD\_CTRL** register, set **IBI\_DATA\_THLD** field.
6. If both PIO Mode and DMA Mode are supported, ensure that the Host Controller is configured to operate in PIO Mode (see *Section 5.3*):
  - A. Write **HC\_CONTROL** register, set **MODE\_SELECTOR** to 1'b1 (**PIO**) (see *Section 7.4.2*).
7. Ensure Host Controller is in running state:
  - A. Read **HC\_CONTROL** register, check if **BUS\_ENABLE** is set to 1'b1 (**ENABLED**).
  - B. Read **HC\_CONTROL** register, check if **RESUME** is set to 1'b0 (**RUNNING**).
  - C. Read **HC\_CONTROL** register, check if **ABORT** is set to 1'b0 (**RUNNING**).
8. Ensure that the PIO Queues are enabled and running:
  - A. Read register **PIO\_CONTROL**, check whether field **ENABLE** is set to 1'b1 (**ENABLED**).
  - B. To start the PIO Queues, set bit field **RS** in register **PIO\_CONTROL**.

### 6.5.2 Command Queue Operation

While the PIO Queues are enabled and running, the Host Controller Driver shall check for available entries on the Command Queue and, for a write transfer, available space on Tx Data Queue. The Tx Data Queue might not be able to fit an entire transfer. The Driver shall populate the Tx Data Queue with data until the Tx Data Buffer threshold condition is not met (i.e., fewer entries than the threshold), preferably in threshold-size batches. Once the Tx Queue is populated (or once all data is posted to the Tx Data Queue), the Driver shall populate the Command Queue with a Command Descriptor. If not all Tx Data for the transfer was posted to the Tx Data Queue, then the interrupt shall indicate to software to load the next batch of data (or threshold batch size). On completion of the transfer, the Response Queue shall be populated with a Response Descriptor per *Section 6.5*. The interrupt indicating successful transfer, and/or transfer errors, shall be flagged in the PIO Interrupt Status register. The software shall read the Response Queue and clear the interrupt.

For read requests, the data may flow before the Response Queue is populated with the Response Status, as it is populated after the Rx Data. If insufficient space is available on the Rx Data Queue, then the Driver shall service the interrupt by reading a threshold-size batch of data from the Rx Data Buffer and clearing the interrupt.

There shall be a 1:1 mapping between the Command Queue's Command Descriptor structures and the Response Queue's Response Descriptor structures, unless the Command Descriptor's **WROC** field indicates that the Host Controller should not generate (i.e., should not provide) a Response Descriptor for successful Write-type transfers.

1643

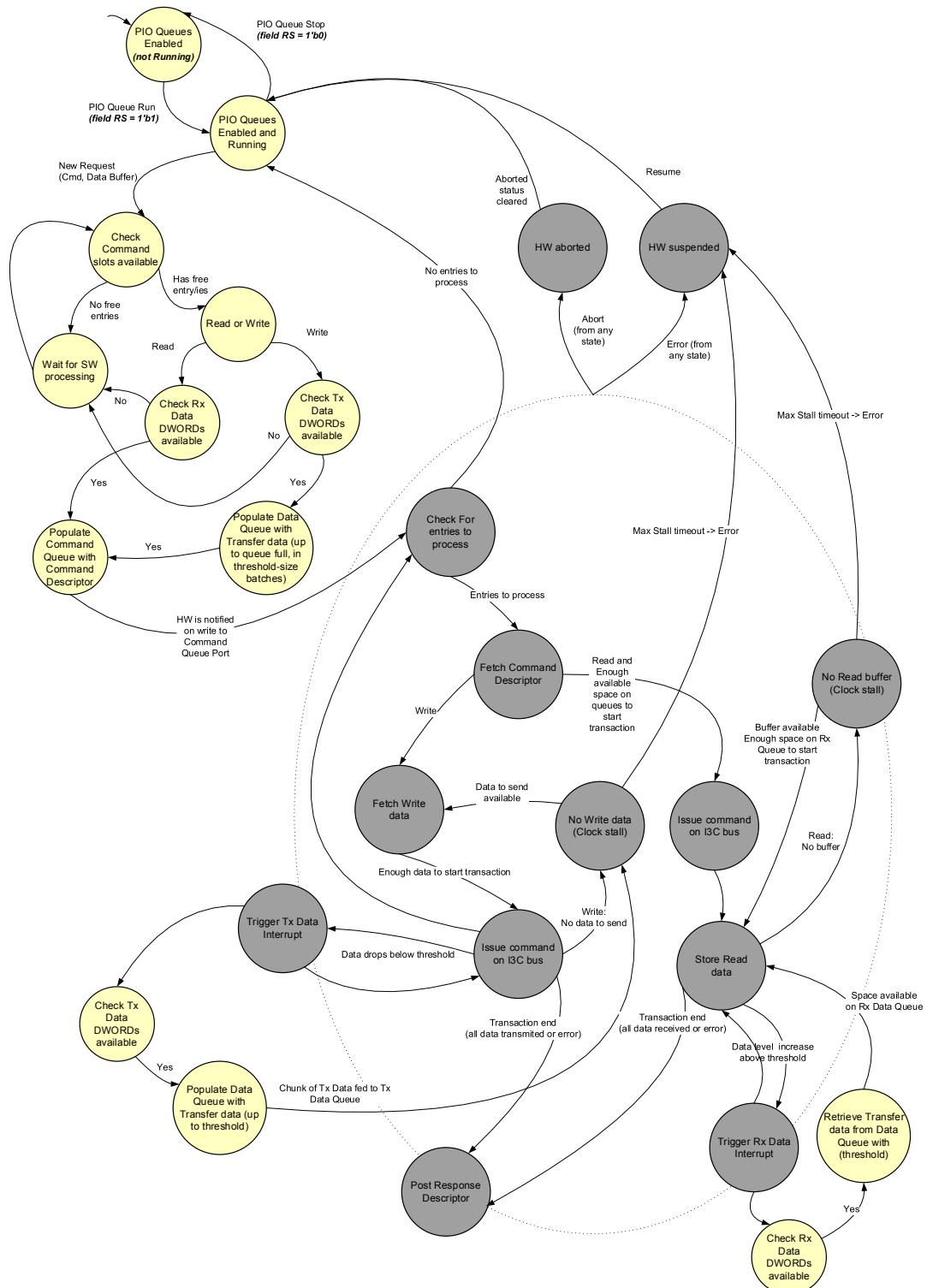


Figure 12 Command Queue FSM

1644

### 6.5.3 Response Queue Operation

While the PIO Queues are enabled and running, upon completion of a Bus transaction and availability of its response status, the Host Controller shall appropriately populate a Response Descriptor structure on the Response Queue as required:

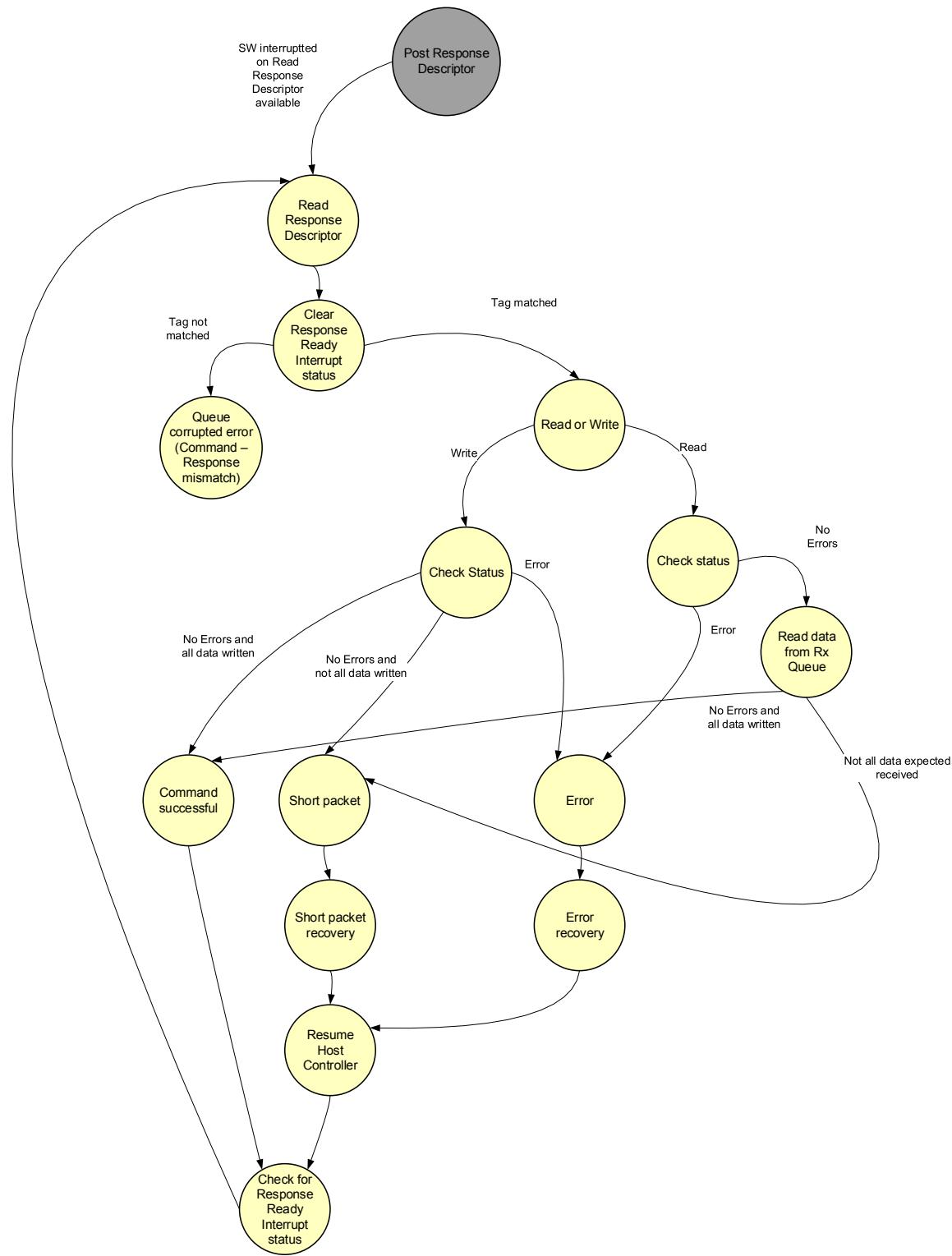
- For successful transfers, this shall be determined by field **WROC** in the corresponding Command Descriptor as well as the transfer type (i.e., Write-type vs. Read-type) (see *Section 6.5*).
- For transfers that generate an error, the Host Controller shall always populate a Response Descriptor.

The Host Controller shall conditionally flag an interrupt to the Driver, depending on the interrupt threshold for the Response Queue (i.e., the value of field **RESP\_BUF\_THLD** in register **QUEUE\_THLD\_CTRL**). Note that the interrupt is subject to both masking and signal enabling, per *Section 6.14*.

Upon processing Response Descriptor entries in the Response Queue, the Driver should observe the change to Response Ready Status (i.e., the de-assertion of interrupt **RESP\_READY\_STAT** in register **PIO\_INTR\_STATUS**) to allow for the Host Controller to flag another interrupt when new Response Descriptors become available, subject to the threshold configuration. This allows for the Driver to process more than a batch of Response Descriptors, based on the threshold configuration; in this manner, the Driver may optionally process a batch in one run, but may choose not to drain the Response Queue fully for certain use cases.

Any read from Response Queue shall evict (i.e., shall dequeue) the Response Descriptor from Response Queue. If the Driver attempts to read from an empty Response Queue, this yields undefined behavior.

1663



**Figure 13 Response Queue FSM**

1664  
1665

### 6.5.4 IBI Queue Operation

For PIO Mode, IBI handling is performed with a single IBI Queue. This IBI Queue is active when the PIO Queues are enabled (i.e., when field **ENABLE** in register **PIO\_CONTROL** is set to 1'b1), meaning that the Bus Controller Logic will accept (i.e., will ACK) incoming IBIs unless the IBI Queue becomes full.

**Note:**

*If the PIO Queues are not enabled (i.e., if field **ENABLE** in register **PIO\_CONTROL** is set to 1'b0), then the Bus Controller Logic will not accept (i.e., will NACK) any incoming IBIs.*

The IBI Queue supports either up to 255 DWORDS of data, or (if extended IBI Queue sizes are supported, per register **ALT\_QUEUE\_SIZE**; see Section 7.5.8) up to 2040 DWORDS of data. The IBI Queue stores IBI Status Descriptor structures (see **Section 8.6**), each of which describes how much IBI Payload data follows it. The payload data is padded with '0's to fill the DWORD entries as appropriate.

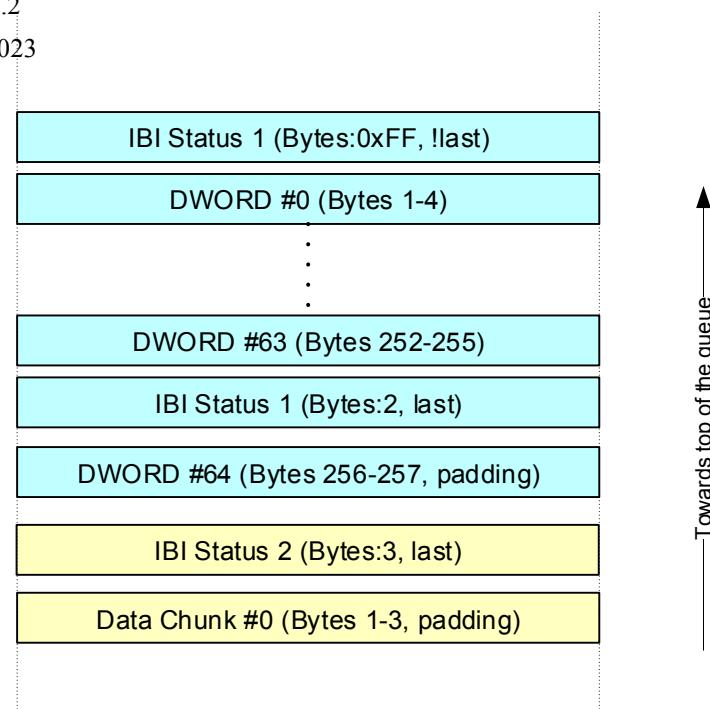
A single IBI event can use multiple IBI Status Descriptor structures, depending on the amount of memory needed to store the data associated with the IBI event. There may be up to 255 Bytes associated with each IBI Status Descriptor. An IBI with less than 256 Bytes of data will require one IBI Status Descriptor structure; an IBI event with 256 Bytes of data will require at least two IBI Status Descriptor structures; and so on. The IBI Status Descriptor structure includes a **LAST\_STATUS** bit, which must be set to 1'b1 for the last IBI Status Descriptor for a given IBI event. To continue an IBI event into an additional IBI Status Descriptor (i.e., for additional storage), set bit field **LAST\_STATUS** to 1'b0.

**Note:**

*Per **Section 6.9.4**, the Driver may also abort an IBI data payload, if the Host Controller supports this capability.*

The IBI Queue has the capability to issue an interrupt when a specific threshold of data or IBI Statuses are reached.

**Example:** *Figure 14* shows an IBI payload that exceeds 255 Bytes, so a second IBI Status Descriptor is used to store the additional received IBI packet data. IBI #1 (blue) contains 257 data Bytes and therefore requires two IBI Status Descriptor entries, whereas IBI #2 (yellow) only needs one IBI Status Descriptor as it only contains three data bytes.



*Legend, color coding*

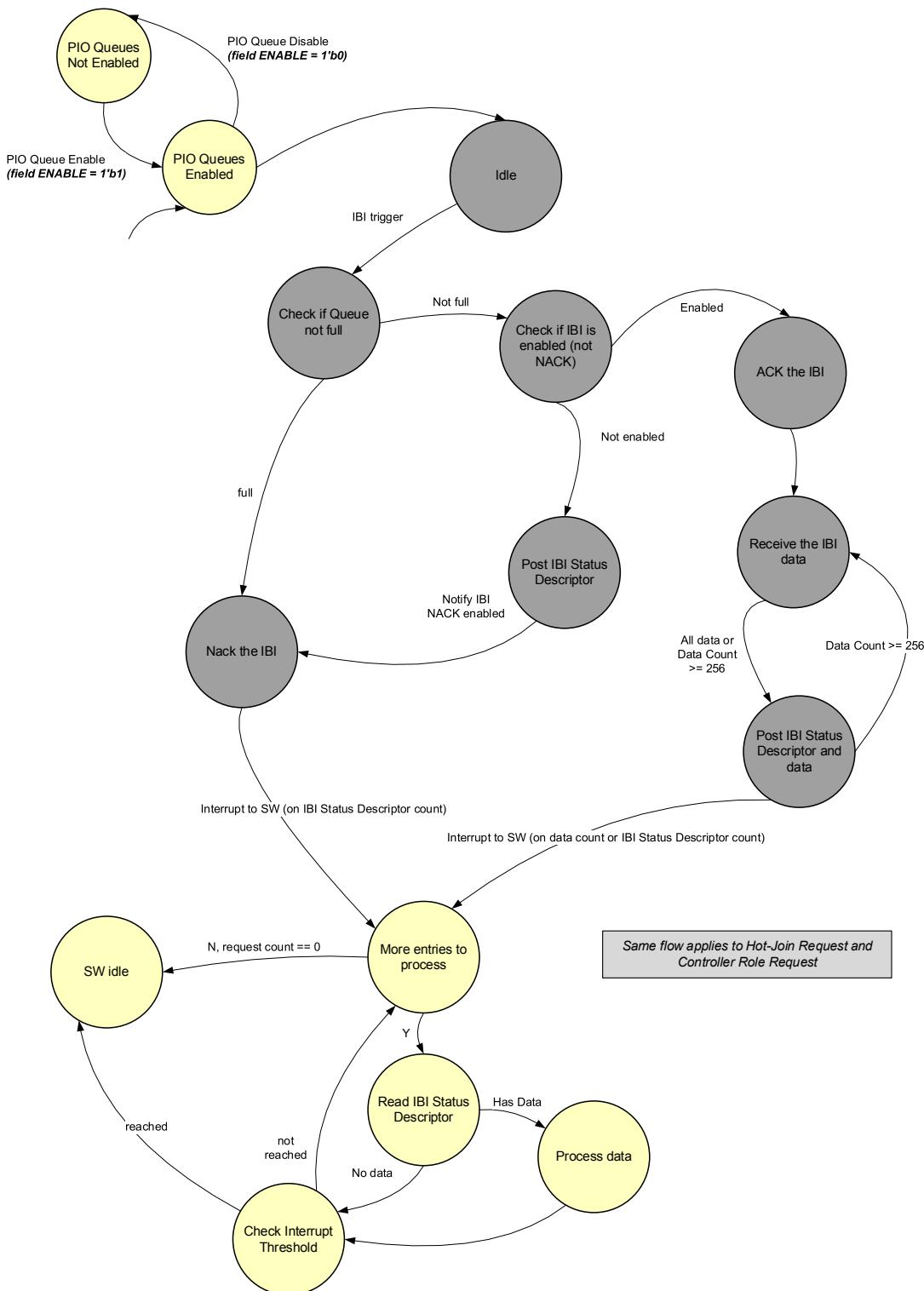
*Structures describing first IBI (large payload IBI example)*

*Structures describing second IBI (regular payload IBI example)*

1692

**Figure 14 IBI Queue Operation for Large and Small Data IBIs**

1693

**Figure 15** illustrates the FSM for the IBI Queue.

1694

**Figure 15 IBI Queue FSM**

### 6.5.5 PIO Thresholds

PIO Mode includes configurable PIO thresholds, which allows the Host Controller and controlling software to use the PIO Queue registers to process Transfer Commands either one at a time, or in batches. The Queue Threshold Control register (i.e., register **QUEUE\_THLD\_CTRL**, see *Section 7.5.5*) and the Transfer Data Buffer Threshold Control register (i.e., register **DATA\_BUFFER\_THLD\_CTRL**, see *Section 7.5.6*) provide configurable threshold settings, to allow software to set interrupt triggers for the following events and conditions:

- A batch of new Commands may be written to the Command Queue
- A batch of Responses is pending processing in the Response Queue
- A batch of IBIs and any related payload data are pending in the IBI Queue
- A batch of Read transfer related data is pending processing in the Rx Data Queue
- A batch of Write transfer related may be written to the Tx Data Queue

The definition allows for setting the size of any batch, to balance latency and performance. For some use cases, the batch size might be decreased with lower threshold values in order to reduce latency for time-critical applications, where controlling delay of processing is required. For other use cases, the batch size might be increased with higher threshold values, which allows for Host system configurations where higher latency might be tolerated.

The Transfer Data Buffer Threshold Control register (i.e., register **DATA\_BUFFER\_THLD\_CTRL**, see *Section 7.5.6*) allows software to configure the appropriate balance of latency and performance for outgoing Write-Type transfers using the Tx Data Queue, as well as incoming Read-Type transfers using the Rx Data Queue.

- The Transmit Buffer Threshold (i.e., field **TX\_BUF\_THLD**) is set by software. The Host Controller shall send an interrupt of type **TX\_THLD\_STAT** when the number of available entries (i.e., DWORDs) in the Tx Data Queue reaches or exceeds the configured threshold level.
  - This interrupt is useful during Write-Type transfers, when not all data DWORDs might initially be enqueued into the Tx Data Queue at the start of the transfer. In such cases, software may need to be notified at the appropriate time to write additional data DWORDs, in order to prevent an underflow condition, which would cause early transaction termination (see *Section 6.8.1*).
  - Upon sending this interrupt, the Host Controller asserts that software may write the indicated number of data DWORDs into the Tx Data Queue (i.e., based on the configured threshold level, per *Section 7.5.9*). As software writes data DWORDs into the Tx Data Queue, this interrupt shall be automatically de-asserted once the Tx Data Queue is filled (i.e., once the number of available entries becomes less than the configured threshold level).
- The Receive Buffer Threshold (i.e., field **RX\_BUF\_THLD**) is set by software. The Host Controller shall send an interrupt of type **RX\_THLD\_STAT** when the number of entries (i.e., DWORDs) in the Rx Data Queue reaches or exceeds the configured threshold level.
  - This interrupt is useful during Read-Type transfers, when data DWORDs are written into the Rx Data Queue during the transfer. In such cases, software may need to be notified at the appropriate time to read (i.e., to consume) the data DWORDs, in order to prevent an overflow condition, which would cause early transaction termination (see *Section 6.8.1*).
  - Upon sending this interrupt, the Host Controller asserts that software may read the indicated number of data DWORDs from the Rx Data Queue (i.e., based on the configured threshold level, per *Section 7.5.9*). As software reads data DWORDs from the Rx Data Queue, this interrupt shall be automatically de-asserted once the Rx Data Queue is drained (i.e., once the number of populated entries becomes less than the configured threshold level).

In addition to the thresholds listed above, it is possible to set the Transfer Start thresholds, in the Transfer Data Buffer Threshold Control register (i.e., register **DATA\_BUFFER\_THLD\_CTRL**, see [Section 7.5.6](#)). Using this register, software may set these thresholds to configure a minimum number of bytes that must be present in the Tx Data Queue to initiate the next enqueued Write-Type transfer; and a minimum number of free entries that must be available in the Rx Data Queue to initiate the next enqueued Read-Type transfer. This method allows for postponing the next enqueued Transfer Command until the indicated Data Queue has reached an appropriate level (i.e., a margin of safety), in order to avoid PIO Queue overrun/underrun conditions that would otherwise cause a stall or timeout. These starting thresholds can be changed by software as needed, depending on the current Data Queue levels, changing system latency, and the size of enqueued Transfer Commands.

- The Transmit Start Threshold (i.e., field **TX\_START\_THLD**) is set by software. It applies when not all data DWORDs have been enqueued into the Tx Data Queue to initiate a Write-Type transfer, when the Host Controller begins processing this transfer as next enqueued Command Descriptor (i.e., the head of the Command Queue).
  - If the next enqueued Command Descriptor is an Immediate Data Transfer Command, or a Regular Data Transfer Command that has a zero-byte payload (i.e., does not rely on the Tx Data Queue), then the Transmit Start Threshold is not considered and the Transfer Command is always allowed to proceed.
  - If the number of data DWORDs that have been written into the Tx Data Queue is sufficient to satisfy the number of bytes indicated by the next enqueued Command Descriptor, then the Transmit Start Threshold is met, so the Transfer Command shall be allowed to proceed.
  - If the number of data DWORDs that have been written into the Tx Data Queue is not sufficient to satisfy the number of bytes indicated by the next enqueued Command Descriptor, then the Transmit Start Threshold is not met, so the Transfer Command shall be postponed until software writes more data DWORDs into the Tx Data Queue.
    - If software subsequently writes additional data DWORDs to meet this threshold condition, then the Transfer Command shall be allowed to proceed.
    - Alternately, if software changes the Transfer Start Threshold to a lower value that causes the threshold condition to be met earlier, then the Transfer Command shall be allowed to proceed.
- The Receive Start Threshold (i.e., field **RX\_START\_THLD**) is set by software. It applies when there are not enough available DWORD entries in the Rx Data Queue to hold data that would be received after initiating a Read-Type transfer, when the Host Controller begins processing this transfer as the next enqueued Command Descriptor (i.e., the head of the Command Queue).
  - If the number of available DWORD entries in the Rx Data Queue is sufficient to satisfy the number of bytes indicated by the next enqueued Command Descriptor, then the Receive Start Threshold shall be met, so the Transfer Command shall be allowed to proceed.
  - If the number of available DWORD entries in the Rx Data Queue is not sufficient to satisfy the number of bytes indicated by the next enqueued Command Descriptor, then the Receive Start Threshold is not met, so the Transfer Command shall be postponed until software reads data DWORDs from the Rx Data Queue.
    - If software subsequently reads these data DWORDs (i.e., from prior Read-Type transfers) from the Rx Data Queue and causes it to drain, then the threshold condition would eventually be met, and the Transfer Command shall be allowed to proceed at that time.
    - Alternately, if software changes the Receive Start Threshold to a lower value that causes the threshold condition to be met earlier, then the Transfer Command shall be allowed to proceed.
  - The Host Controller's ability to notify software of data DWORDs in the Rx Data Queue from prior Read-Type transfers shall depend on the current value of the Receive Buffer Threshold (i.e., field **RX\_BUF\_THLD**), so software must provide values for both Receive thresholds that appropriately balance performance and latency of the system. Software must manage frequent Read-Type transfers properly, and prevent stall conditions that turn into aborted transactions, if the Rx Data Queue becomes full during a transfer and is not drained in a timely manner.

1790

**Note:**

1791

*In PIO Mode, if the Host Controller must postpone a Transfer Command due to either of the starting thresholds not being met, then all enqueued Transfer Commands are blocked from processing until the threshold condition is met, or until software reduces the starting threshold (if possible).*

1792

*The Transmit Start Threshold may be set to a minimum value of 2 DWORDs, whereas the smallest possible Write-Type transfer that relies on the Tx Data Queue might only use 1 DWORD entry (i.e., between 1 and 4 data bytes). For such small Write-Type transfers, the threshold would always be met regardless of the value written to field TX\_START\_THLD, since the single data DWORD written into the Tx Data Queue would always satisfy the starting condition. The Host Controller shall compare the length of the transfer in the Command Descriptor (i.e., field DATA\_LENGTH in bytes) with the number of DWORD entries currently written, and handle such short Write-Type transfers by allowing them to proceed if the Tx Data Queue contains enough DWORD entries based on the length of the transfer.*

1793

1794

1795

1796

1797

1798

1799

1800

1801

1802

### 6.5.6 PIO Abort Operation

The Driver may trigger the PIO Abort operation for the PIO Queues by setting the **ABORT** bit in register **PIO\_CONTROL** (*Section 7.5.12*) to 1'b1.

If the PIO Queues are enabled and running, then:

- The Host Controller shall at the earliest opportunity abort (i.e., terminate) any in-progress transfers for the PIO Queues that the I3C Bus Controller Logic is currently driving.

In terminating transfers, the I3C Bus Controller Logic shall observe all requirements of the I3C Specification [**MIPI02**] for the current I3C Mode (i.e., shall not cause any I3C Bus errors).

The transactions that were aborted shall be indicated accordingly in the corresponding Response Descriptors; field **ERR\_STATUS** shall contain 0x8 (**HC\_ABORTED**). The Host Controller shall also trigger the **TRANSFER\_ABORT\_STAT** interrupt (if enabled for notification) via register **PIO\_INTR\_STATUS** (see *Section 7.5.9*).

**Note:**

*If the Host Controller is instructed to cancel a transaction sequence (i.e., Command Descriptor with field **TOC**=0) in order to process the PIO Abort operation, then it shall also trigger an interrupt via field **HC\_SEQ\_CANCEL\_STAT** in register **INTR\_STATUS** (*Section 7.4.7*) to inform the Host that the sequence was terminated before processing a Command Descriptor with field **TOC**=1 (see *Section 6.12*).*

If the PIO Queues are enabled and running but not currently active for processing, or if the PIO Queues were enabled but not currently running, then no transactions shall be aborted (because the I3C Bus Controller Logic was not actively processing any Command Descriptors).

**Note:**

*Any subsequent Command Descriptors that might have been enqueued after any aborted transfers shall remain unprocessed in the Command Queue once the PIO Queues enter the **ABORTED** state. The Driver should determine whether such Command Descriptors should remain enqueued for future processing, or whether the Command Queue should be flushed (i.e., via register **RESET\_CONTROL**) before the Driver enqueues new Command Descriptors.*

To resume operation for the PIO Queues, the Driver shall perform the following steps:

1. If any transfers were aborted (as specified above), then clear the Transfer Abort Status interrupt notification for the PIO Queues by writing 1'b1 to field **TRANSFER\_ABORT\_STAT** in register **PIO\_INTR\_STATUS** (*Section 7.5.9*).
2. Clear the PIO Abort Request state by writing 1'b0 to field **ABORT** in register **PIO\_CONTROL** (*Section 7.5.13*).
3. Re-enable processing of PIO Queues by writing 1'b1 to field **RS** in register **PIO\_CONTROL**. Per *Section 6.8.1*, this resumes PIO Queue operation for Command Descriptors and Response Descriptors, unless the PIO Queues entered the **ABORTED** state via the Host Controller Abort operation (see *Section 6.8.4*) or unless the Host Controller remains in **ABORTED** state.

**Note:**

*A PIO Abort can also be triggered by a global Host Controller Abort operation (see *Section 6.8.4*). If this occurs, then both register **HC\_CONTROL** (i.e., at the global level) and register **PIO\_CONTROL** shall report the **ABORTED** state. However, if only a PIO Abort was triggered on its own (i.e., via register **PIO\_CONTROL**), then this does not necessarily cause, and does not necessarily imply, a global Host Controller Abort operation.*

## 6.6 Ring Management

**Note:**

This section applies only for Host Controller implementations that implement DMA Mode, and thus connect to a System Bus that supports memory-mapped IO and DMA.

In the I3C HCI Register Map, at the Section Offset for Ring Headers Descriptor (see [Section 7.4.13](#)), a metadata area appears containing information on the number of Ring Headers that the Host Controller supports, and the Header structure size (see [Section 7.6.1](#)). Every Header describes a single Ring Bundle, which includes:

- One **Command/Response Ring Pair** (i.e., a Command Ring and a Response Ring) that provide a Command/Response interface to software; and
- One **IBI Ring Pair** (i.e., an IBI Status Ring and an IBI Data Ring) that provide IBI notification status and data.

A Host Controller may support up to 8 Ring Bundles. Each Ring Bundle is described by its own Ring Header (see [Section 7.6.10](#)). Each Ring Header supports up to 255 (a static number) valid Ring entries for Command, Response, and IBI Status Rings.

Every Ring Header includes:

- A **Run/Stop** bit, which stops processing on all Rings described by that Ring Header (see [Section 7.6.10.8](#) and [Section 7.6.10.9](#)).
- An **Enable/Disable** bit, which enables or disables all Rings described by specific Ring Header. A Disable/Enable sequence is used to reset a given Ring Header (see [Section 7.6.10.8](#) and [Section 7.6.10.9](#)).

**Note:**

Support for multiple Rings allows the software to direct separate pipes to different Device groups, which may be useful for some Device classes. For example, each individual pipe may feature different behavior in terms of processing latency.

### 6.6.1 Ring Bundle Initialization

Before a Ring Bundle can be used, the software needs to know how many Ring Headers the specific Host Controller implementation supports, and to provide allocated memory for all Rings that it wishes to use.

- The minimum memory required for the Command Ring and Response Ring shall be derived from the number of entries multiplied by the minimum size of the data structure used for each Ring.

For the Command Ring, the Host Controller may use additional DWORDS for storing context, at the discretion of the implementer.

- The minimum memory required for the IBI Status Ring and IBI Data Ring is arbitrary, and is not related to the Command Ring or Response Ring. This shall be derived from the size of the IBI Status Descriptor structure (see [Section 8.6](#)) and the desired size of the IBI Data Ring.

To initialize a specific Ring Bundle, the Driver shall:

1. Set the sizes of the Command and Response Rings, and set up the IBI Status and IBI Data Ring (optional; see note below).
  - A. In register **CR\_SETUP**, evaluate the sizes of the Transfer Descriptor (**XFER\_STRUCT\_SIZE**) and Response Descriptor (**RESP\_STRUCT\_SIZE**) structures ([Section 7.6.10.1](#)).
  - B. In register **IBI\_SETUP**, evaluate the size of the IBI Status Descriptor structure (**IBI\_STATUS\_STRUCT\_SIZE**, see [Section 7.6.10.2](#)).
  - C. In register **CR\_SETUP**, set the **RING\_SIZE** field ([Section 7.6.10.1](#)).
  - D. In register **IBI\_SETUP**, set fields **IBI\_STATUS\_RING\_SIZE**, **CHUNK\_SIZE**, and **CHUNK\_COUNT** ([Section 7.6.10.2](#)) (optional; see note below).

- 1888    2. Allocate memory for the Rings:
  - 1889    A. The Command Ring size is **XFER\_STRUCT\_SIZE \* RING\_SIZE**
  - 1890    B. The Response Ring size is **RESP\_STRUCT\_SIZE \* RING\_SIZE**
  - 1891    C. The IBI Status Ring size is **IBI\_STATUS\_STRUCT\_SIZE \* IBI\_STATUS\_RING\_SIZE**
  - 1892    D. The IBI Data Ring size is  $(2^{\text{CHUNK\_SIZE}+2} * \text{CHUNK\_COUNT})$
- 1893    3. Enable the Ring Header:
  - 1894    A. To enable the Ring Header, set the **ENABLE** bit in register **RH\_CONTROL** (*Section 7.6.10.9*).
- 1895    4. Upon Ring Header being enabled, the Host Controller shall zero the Host-Controller-managed fields in  
1896    the following registers:
  - 1897    • **RH\_OPERATION1** (*Section 7.6.10.10*)
  - 1898    • **RH\_OPERATION2** (*Section 7.6.10.11*)
  - 1899    • **CHUNK\_CONTROL** (*Section 7.6.10.3*)
- 1900    5. Set up the memory pointer registers for each Ring:

1901    **Note:**

1902    *The memory allocated for each Ring must be physically contiguous, unless Scatter-Gather is*  
1903    *supported and enabled for each Ring Pair (see **Section 6.2.1**).*

  - 1904    A. For the Command Ring, set registers **RH\_CMD\_RING\_BASE\_LO** and **RH\_CMD\_RING\_BASE\_HI**  
*(Section 7.6.10.12 and Section 7.6.10.13)*.
  - 1906    B. If field **SG\_CAPABILITY\_CR\_EN** in register **HC\_CAPABILITIES** is 1'b1 (**ENABLED** per *Section 7.4.4*):

1907       Optionally configure the Command Ring for Scatter-Gather:  
1908       Set register **RH\_CMD\_RING\_SG** (*Section 7.6.10.20*) to accept a non-contiguous  
1909       memory allocation (i.e., an array of Memory Descriptors per *Section 8.7*).
  - 1910    C. For the Response Ring, set registers **RH\_RESP\_RING\_BASE\_LO** and **RH\_RESP\_RING\_BASE\_HI**  
*(Section 7.6.10.14 and Section 7.6.10.15)*.
  - 1912    D. If field **SG\_CAPABILITY\_CR\_EN** in register **HC\_CAPABILITIES** is 1'b1 (**ENABLED** per *Section 7.4.4*):

1913       Optionally configure the Response Ring for Scatter-Gather:  
1914       Set register **RH\_RESP\_RING\_SG** (*Section 7.6.10.21*) to accept a non-contiguous  
1915       memory allocation (i.e., an array of Memory Descriptors per *Section 8.7*).
  - 1916    E. For the IBI Status Ring, set registers **RH\_IBI\_STATUS\_RING\_BASE\_LO** and  
**RH\_IBI\_STATUS\_RING\_BASE\_HI** (*Section 7.6.10.16* and *Section 7.6.10.17*).
  - 1918    F. If field **SG\_CAPABILITY\_IBI\_EN** in register **HC\_CAPABILITIES** is 1'b1 (**ENABLED** per *Section 7.4.4*):

1919       Optionally configure the IBI Status Ring for Scatter-Gather:  
1920       Set register **RH\_IBI\_STATUS\_RING\_SG** (*Section 7.6.10.22*) to accept a  
1921       non-contiguous memory allocation (i.e., an array of Memory Descriptors per  
1922       *Section 8.7*).
  - 1923    G. For the IBI Data Ring set registers **RH\_IBI\_DATA\_RING\_BASE\_LO** and **RH\_IBI\_DATA\_RING\_BASE\_HI**  
*(Section 7.6.10.18 and Section 7.6.10.19)*.
  - 1925    H. If field **SG\_CAPABILITY\_IBI\_EN** in register **HC\_CAPABILITIES** is 1'b1 (**ENABLED** per *Section 7.4.4*):

1926       Optionally configure the IBI Data Ring for Scatter-Gather:  
1927       Set register **RH\_IBI\_DATA\_RING\_SG** (*Section 7.6.10.23*) to accept a  
1928       non-contiguous memory allocation (i.e., an array of Memory Descriptors per  
1929       *Section 8.7*).
- 1930    6. Zero the Host Controller Driver-managed fields (R/W) in the following registers:
  - 1931    • **RH\_OPERATION1** (*Section 7.6.10.10*)
  - 1932    • **RH\_OPERATION2** (*Section 7.6.10.11*)
  - 1933    • **CHUNK\_CONTROL** (*Section 7.6.10.3*)

- 1934 7. Start the Ring Header:
- 1935 A. To enable interrupts, set registers **RH\_INTR\_STATUS\_ENABLE** and **RH\_INTR\_SIGNAL\_ENABLE**  
1936 (*Section 7.6.10.5* and *Section 7.6.10.6*).
- 1937 B. To start the Ring Header, set the RS bit in register **RH\_CONTROL** (*Section 7.6.10.9*).

1938 **Note:**

1939 *Under normal operating conditions, at least one Ring Bundle should typically have its IBI Ring Pair*  
1940 *configured for use (i.e., initialized and enabled as per the steps above). Otherwise, the Driver will*  
1941 *not be able to receive any IBI Requests. The Driver may choose to not use the IBI Ring Pair for a*  
1942 *particular Ring Bundle, at the discretion of the Driver developer or the particular Host system*  
1943 *configuration. In such cases, the IBI Ring Pair might not be enabled. However, the IBI Ring Pair for*  
1944 *Ring Bundle 0 should generally be initialized and enabled, as Ring Bundle 0's IBI Ring Pair is used*  
1945 *for special purposes such as Hot-Join Requests and IBI Requests from any I3C Target Devices that*  
1946 *do not have a dedicated DAT entry (see **Section 6.9.2**). As noted in **Section 5.4**, Ring Bundle 0's*  
1947 *IBI Ring Pair must be initialized and enabled for the Driver to be able to receive these requests.*

### 6.6.2 Command / Response Ring Pairs

1948 Each Ring Bundle includes a Command / Response Ring Pair, consisting of one Command Ring and one  
 1949 Response Ring.

1950 As **Table 4** details, an Enqueue Pointer, a Dequeue Pointer, and a Software Dequeue Pointer are used for  
 1951 both Rings in a Command / Response Ring Pair.

**Table 4 Pointers for Command / Response Ring Pair**

Purpose	Register Field	Ring Header Register	Updated by
Enqueue Pointer Command Ring	CR_ENQ_PTR	RH_OPERATION1 see <b>Section 7.6.10.10</b>	Driver Command Ring producer Notify on write as 'Doorbell'
Dequeue Pointer Command / Response Ring	CR_DEQ_PTR	RH_OPERATION2 see <b>Section 7.6.10.11</b>	Host Controller Command Ring consumer Response Ring producer
Software Dequeue Pointer Response Ring	CR_SW_DEQ_PTR	RH_OPERATION1 see <b>Section 7.6.10.10</b>	Driver Response Ring consumer

1952 **Note:**

1953     *The Dequeue Pointer serves two purposes, since the Host Controller must simultaneously process  
 1954       any enqueued Transfer Descriptors (from the Command Ring) and populate Response Descriptors  
 1955       (into the Response Ring) for each such Transfer Descriptor, while the Ring Bundle is enabled and  
 1956       running. See **Section 6.6.2.1** and **Section 6.6.2.2**.*

1957 Within a Command/Response Ring Pair, the Command and Response Rings always have the same length  
 1958 (number of structures). The Command Ring contains Transfer Descriptor structures (see **Section 8.3**), and  
 1959 the Response Ring contains Response Descriptor structures (see **Section 8.5**).

1960 Each Command/Response Ring Pair serves as an independent interface to software. If the Ring Bundle is  
 1961 enabled, then Transfer Commands (i.e., Command Descriptors contained within Transfer Descriptors) that  
 1962 are enqueued by software into the Command Ring would be processed by the Host Controller, and the  
 1963 corresponding results for each Transfer Command (i.e., Response Descriptors) would be written into the  
 1964 Response Ring contained in the same Ring Bundle.

1965 In addition to the ability to gracefully stop execution by software request, transfers enqueued in a  
 1966 Command Ring Pair may also be aborted upon detection of any malfunction. In this case, any transfer  
 1967 currently executing on the Bus may be interrupted (i.e., may be completed with an error condition) and the  
 1968 Host Controller shall then stop the Command/Response Ring Pair, and wait for recovery by the software.

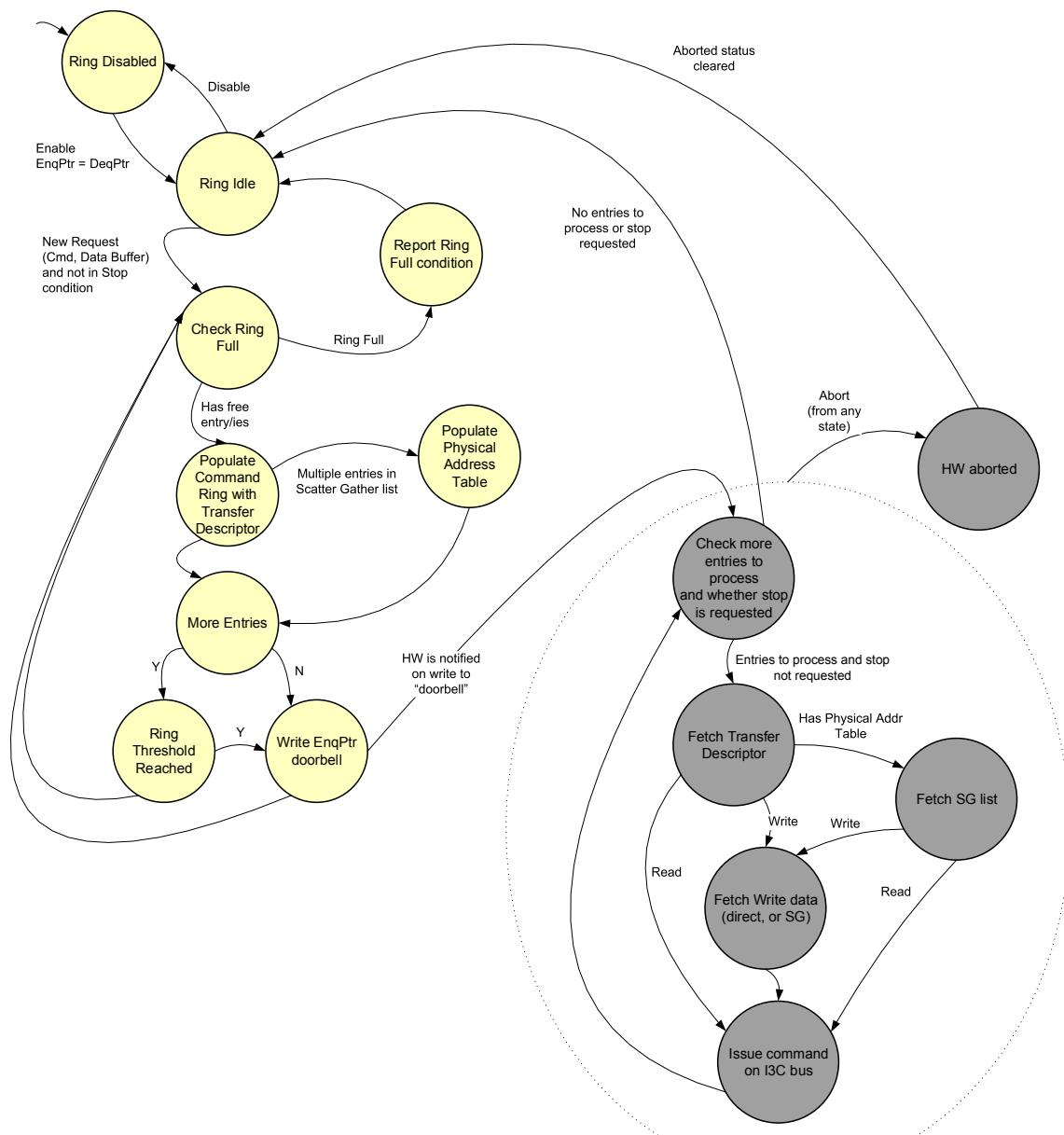
- 1969     • If the Host Controller detects a transfer error, then it shall stop processing and transition the Ring Bundle  
 1970       into **Stopped** status. The Host Controller shall also indicate that a transfer error has occurred, by asserting  
 1971       the **TRANSFER\_ERR\_STAT** interrupt (if enabled for notification) for the associated Ring Header (see  
 1972       **Section 7.6.10.4**).
- 1973     • If software changes the Ring Bundle into **Running** or **Stopped** status, then the Host Controller shall also  
 1974       indicate that the Ring Bundle's state has changed by asserting the **RING\_OP\_STAT** interrupt (if enabled for  
 1975       notification) for the associated Ring Header (see **Section 7.6.10.4**).
- 1976     • Software may also initiate a Ring Abort operation per **Section 6.6.6**.

### 6.6.2.1 Command Ring Operation

The Host Controller Driver shall check for available entries on the Command Ring for an already enabled and started Ring Bundle, and shall schedule transfer(s) by putting Transfer Descriptor(s) (see *Section 8.3*) onto the Command Ring, and then shall update the Enqueue Pointer (**EnqPtr**), which shall serve as a Doorbell to the Host Controller. Upon detecting a Doorbell ring, the Host Controller shall check the Enqueue Pointer and shall process Transfer Descriptors in the Command Ring, in order, until reaching the Enqueue Pointer (and no further) (see *Section 7.6.10.10* and *Section 7.6.10.11*). For each processed Transfer Descriptor, upon completion of the corresponding transaction on the Bus, the Host Controller shall populate a Response Descriptor (see *Section 8.5*) structure in the Response Ring.

There shall be a 1:1 mapping between the Command Ring's Transfer Descriptor structures and the Response Ring's Response Descriptor structures. As a result, the Ring management related registers for a given Ring Pair are shared between these two Rings.

Transfers shall be split according to the width of the **DATA\_LENGTH** field. E.g., for a 16-bit **DATA\_LENGTH** field a single Transfer Descriptor can describe a transfer of up to 65535 Bytes (0xFFFFh). If the transfer is larger than that, then it should be split into multiple transfers, each with a maximum length of 65535 Bytes.



1991

**Figure 16 Command Ring FSM**

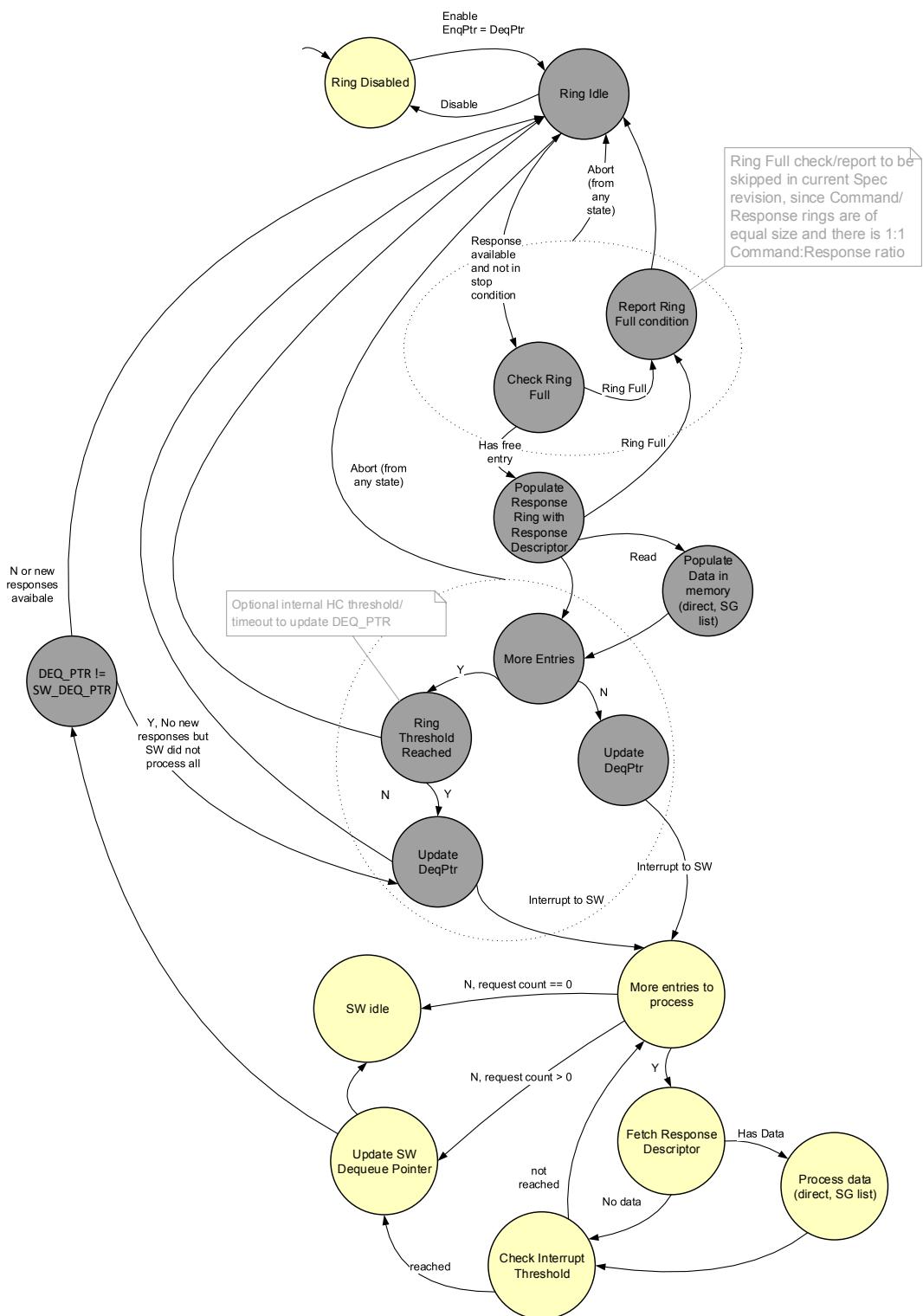
### 6.6.2.2 Response Ring Operation

Upon completion of a Bus transaction and availability of its response status, the Host Controller shall populate a Response Descriptor structure (see *Section 8.5*) on the Response Ring, update Dequeue Pointer (**DeqPtr**), and optionally trigger a **TRANSFER\_COMPLETION\_STAT** interrupt to the Driver, depending on the value of the Transfer Descriptor's **IOC** bit. Note that the interrupt is subject to both masking and signal enabling. If enabled, this interrupt shall appear in register **RH\_INTR\_STATUS** (see *Section 7.6.10.4*) in the Ring Header for the associated Ring Bundle. Upon successful completion of the transaction, the Host Controller shall populate the Response Descriptor's **ERR\_STATUS** field with the value 0x0.

The list position of the newly populated Response Descriptor in the Response Ring shall be the same as the list position of the Transfer Descriptor that caused the transfer (or command) in the Command Ring.

**Example:** If a Bus transfer is caused by the third Transfer Descriptor in the Command Ring, then the Response Descriptor that reports the response to that transfer shall appear in the third position in the Response Ring.

Upon processing Response Descriptor entries in the Response Ring, the Driver shall indicate its current processing position in the Response Ring (i.e., shall indicate which Response Descriptor entries in the Response Ring are now available for the Host Controller to populate) by updating the Software Dequeue Pointer (**SwDequeuePointer**) (see *Section 7.6.10.10* and *Section 7.6.10.11*). A write to **SwDequeuePointer** notifies the Host Controller that the software event handler has exited. If any Response Descriptors remain unprocessed (outstanding) at that time, then the Host Controller shall repeat the interrupt to the Driver so that processing of received Responses can continue to completion.



2011

**Figure 17 Response Ring FSM**

### 6.6.3 IBI Ring Pairs

2012 For a given Ring Bundle, IBI handling is shared between the IBI Status Ring and the IBI Data Ring.  
2013 As *Table 5* details, an Enqueue Pointer and a Dequeue Pointer are used for the IBI Status Ring.

**Table 5 Pointers for IBI Ring Pair**

Purpose	Register Field	Ring Header Register	Updated by
<b>Enqueue Pointer</b> IBI Status Ring	<b>IBI_ENQ_PTR</b>	<b>RH_OPERATION2</b> see <i>Section 7.6.10.11</i>	<b>Host Controller</b> Producer
<b>Dequeue Pointer</b> IBI Status Ring	<b>IBI_DEQ_PTR</b>	<b>RH_OPERATION1</b> see <i>Section 7.6.10.10</i>	<b>Driver</b> Consumer Notify on write as ‘Doorbell’

2014 Field **CHUNK\_COUNTER** in register **CHUNK\_CONTROL** (see *Section 7.6.10.3*) indicates the first Data Chunk  
2015 in the IBI Data Ring, for the IBI Status Descriptor at the position where the Dequeue Pointer is pointing  
2016 (see *Section 6.6.3.2*).

2017 The IBI Status Ring supports up to 255 valid IBI Status Descriptor structures (see *Section 8.6*), each  
2018 describing a number of chunks of memory on the IBI Data Ring in the same Ring Bundle.

2019 Each Data buffer consists of one or more equally-sized Data Chunks. This Data Chunk size is variable, and  
2020 is determined by the Driver via field **CHUNK\_SIZE** in register **IBI\_SETUP** (see *Section 7.6.10.2*). For  
2021 example, a 4kB total buffer block could be processed either as 1000 DWORD chunks, or as 500 QWORD  
2022 chunks, etc.

#### 6.6.3.1 IBI Status Ring Operation

2023 A single IBI event can use from one to 255 IBI Status Descriptor structures, depending on the amount of  
2024 memory in the IBI Data Ring needed to store the IBI payload data associated with the IBI event. There may  
2025 be up to 255 Data Chunks associated with each IBI Status Descriptor (see *Section 6.6.3.2*). Note that all  
2026 Data Chunks for an IBI Status Descriptor shall be “full” (i.e., fully populated with IBI payload data), except  
2027 for the last Data Chunk: the Host Controller shall write incoming IBI payload data into all such Data  
2028 Chunks for an IBI Status Descriptor, without gaps, until the end of the IBI payload.

2029 The Host Controller shall automatically create IBI Status Descriptor structures and enqueue them onto the  
2030 IBI Status Ring, based on the overall length of the IBI data payload. An IBI needing one Data Chunk of  
2031 storage (or less, for shorter payloads) will require one IBI Status Descriptor structure. However, an IBI  
2032 event needing 256 Data Chunks of storage will require at least two consecutive IBI Status Descriptor  
2033 structures; and so on. The IBI Status Descriptor structure includes a **LAST\_STATUS** bit, which shall be set to  
2034 1'b1 for the last IBI Status Descriptor for a given IBI event. To continue an IBI event into an additional IBI  
2035 Status Descriptor (i.e., for additional storage), the Host Controller shall set field **LAST\_STATUS** to 1'b0.

2036 **Note:**

2037 Considering the two paragraphs above: For an IBI event that has a longer data payload and  
2038 requires two or more IBI Status Descriptors, the Host Controller should generally write incoming IBI  
2039 payload data to completely fill all Data Chunks associated with the first IBI Status Descriptor.  
2040 Additionally, all Data Chunks associated with each subsequent IBI Status Descriptor should be “full”  
2041 Data Chunks, up until the last IBI Status Descriptor where field **LAST\_STATUS** is set to 1'b1.

2042 The Driver may also abort an IBI data payload, if the Host Controller supports this capability (per  
2043 *Section 6.9.4*).

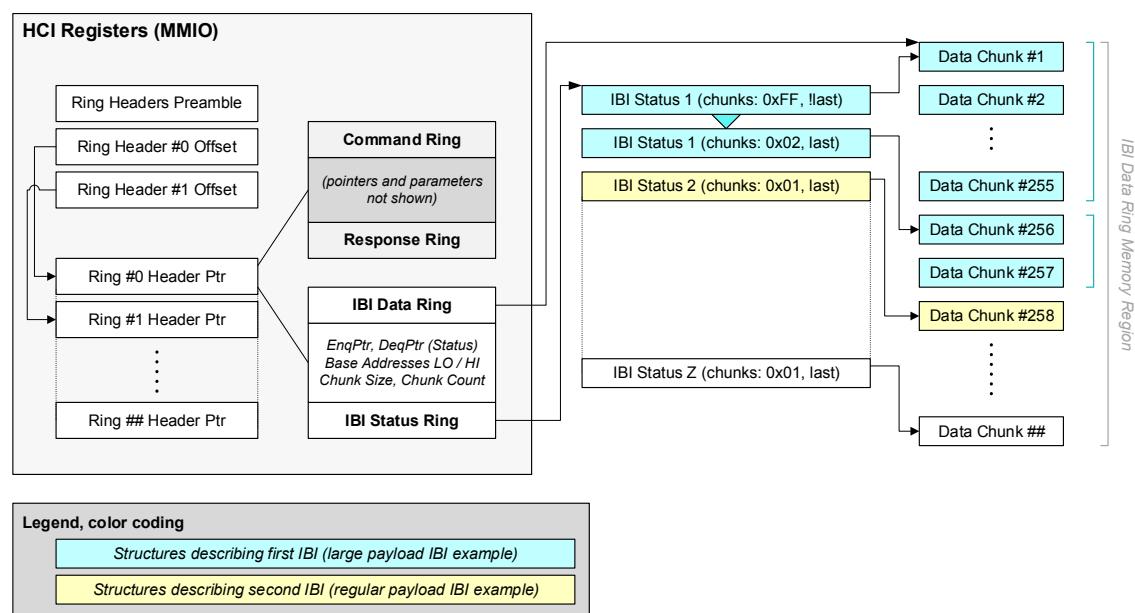
2044 To enqueue one or more IBI Status Descriptor structures for an IBI event, the Host Controller shall  
2045 populate these IBI Status Descriptor structures into the IBI Status Ring, update the Enqueue Pointer, and  
2046 then optionally trigger the **IBI\_READY\_STAT** interrupt to the Driver. Note that this interrupt is subject to both  
2047 masking and signal enabling. If enabled, this interrupt shall appear in register **RH\_INTR\_STATUS** (see

2048     **Section 7.6.10.4)** in the Ring Header for the associated Ring Bundle, to indicate the availability of an IBI  
 2049     Status Descriptor and any related Data Chunk(s) for the Driver to consume from this IBI Ring Pair.

2050     Once the Driver receives this interrupt, it may write to the Dequeue Pointer, which notifies the Host  
 2051     Controller that the Driver has consumed one or more IBI Status Descriptor structures from the IBI Status  
 2052     Ring (see **Section 6.9.2** and **Section 7.6.10.10**).

2053     The Host Controller shall ensure that it does not enqueue any new IBI Status Descriptor structures that  
 2054     might overwrite any structures that have not yet been consumed by the Driver, by first checking the value  
 2055     of the Dequeue Pointer. If the Driver has not yet updated the Dequeue Pointer to advance its processing  
 2056     position in the IBI Status Ring, then the Host Controller shall not add new IBI Status Descriptor structures.  
 2057     However, if the IBI Status Ring is full, or becomes full during an IBI event, and cannot accept new  
 2058     structures until the Driver has advanced the Dequeue Pointer, then the Host Controller shall trigger the  
 2059     **IBI\_RING\_FULL\_STAT** interrupt to the Driver, and must either terminate reception of the IBI data payload  
 2060     (i.e., only accept a partial entry) and mark field **LAST\_STATUS** as 1'b1; or else it must instruct the I3C Bus  
 2061     Controller Logic to stall the Bus (if possible) until the Driver can resolve the situation and make more  
 2062     space in the IBI Status Ring for additional structures. Similar conditions apply if the IBI Data Ring is full  
 2063     upon receiving an IBI, or becomes full during reception of an IBI data payload (see **Section 6.6.3.2**). If the  
 2064     Driver cannot resolve the situation in a timely manner, then the Host Controller must terminate reception of  
 2065     the IBI payload, and must also NACK all incoming IBI events until the Driver can process entries in the  
 2066     IBI Status Ring to make more space for additional structures.

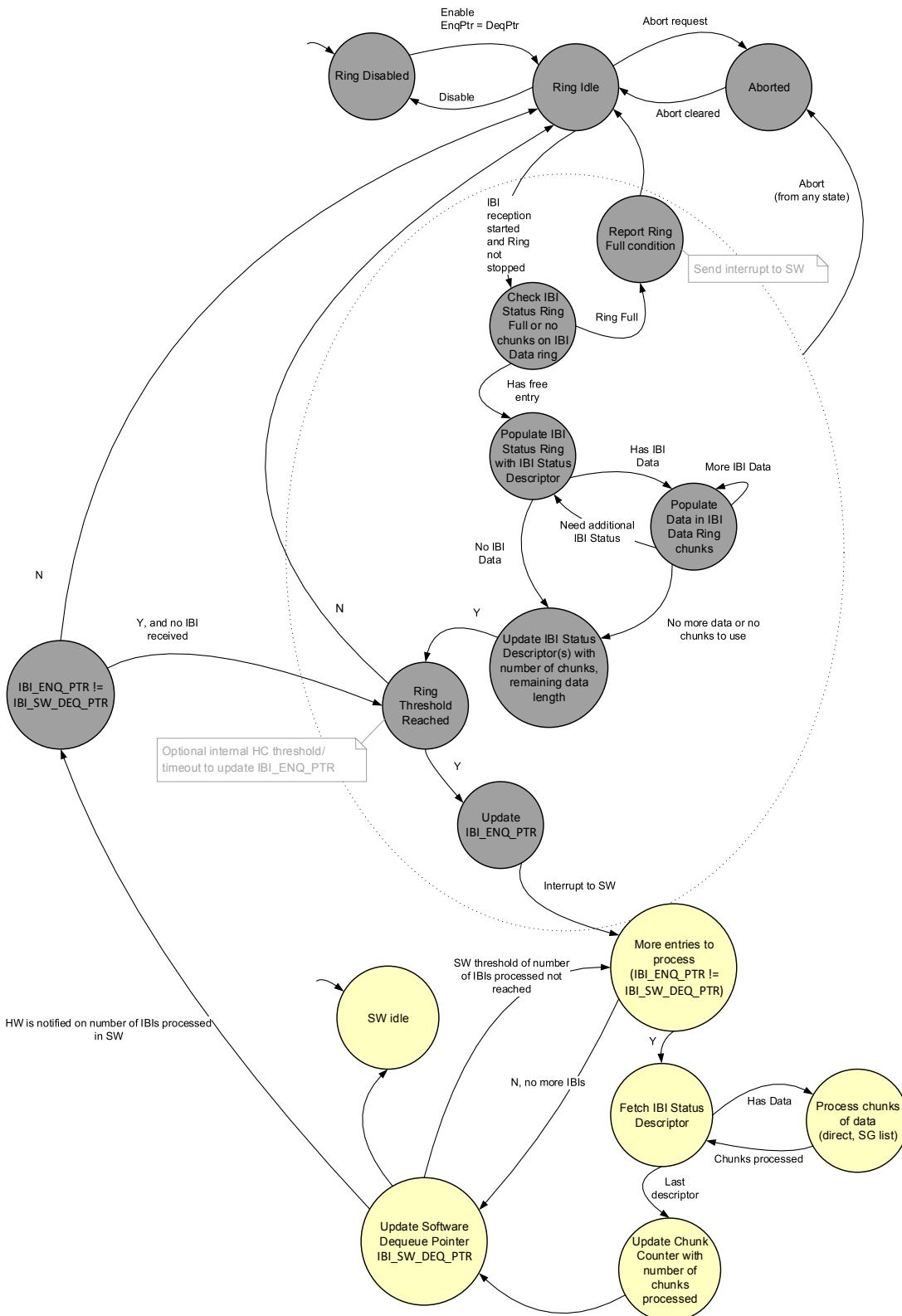
2067     **Example:** *Figure 18* depicts the IBI Status Ring and IBI Data Ring after capturing two IBI events on the  
 2068     I3C Bus. In this example, the IBI Ring Pair have captured an IBI event that exceeds the maximum Chunk  
 2069     count (i.e., 255 Data Chunks per IBI Status Descriptor), so a second IBI Status Descriptor is used to store  
 2070     the additional received IBI payload data. IBI event #1 (which is shaded blue in the figure) consumes 257  
 2071     Data Chunks, and as a result requires two IBI Status Descriptor structures. By contrast, IBI event #2  
 2072     (yellow shading) consumes only one Data Chunk and as a result requires only one IBI Status Descriptor  
 2073     structure.



2074     **Figure 18 IBI Ring Data Chunk Concept**

2075

**Figure 19** illustrates the FSM for IBI Rings.



**Figure 19 IBI Ring FSM**

2076

### 6.6.3.2 IBI Data Ring Operation

A single IBI Status Descriptor may require up to 255 Data Chunks from the IBI Data Ring. Since the Host Controller does not always know the full (i.e., intended) length of the IBI payload when the I3C Bus Controller Logic receives and accepts the IBI, and since the memory regions for the IBI Status Ring and IBI Data Ring are allocated separately by the Driver, the Host Controller shall manage the IBI Ring Pair in an efficient manner, for IBI payloads of varying length.

The Host Controller shall allocate (i.e., shall mark as used) up to 255 Data Chunks as needed for each IBI Status Descriptor, starting at the first free Data Chunk. An allocation of Data Chunks shall either be adjacent (i.e., allocated in a single region) within the IBI Data Ring, or, if the end of the IBI Data Ring is reached, an allocation shall “wrap” around the end of the IBI Data Ring and continue from the start of the IBI Data Ring (i.e., from position 0). The Host Controller may allow an allocation to grow until a maximum of 255 Data Chunks. At the end of an allocation, the Host Controller shall automatically attempt to create a new IBI Status Descriptor, per *Section 6.6.3.1*, if the IBI has not been terminated.

If the IBI Data Ring becomes completely full during an IBI event, then the Host Controller shall trigger the **IBI\_RING\_FULL\_STAT** interrupt, and shall either terminate reception of the IBI payload (i.e., only accept a partial entry for this IBI Status Descriptor and mark its field **LAST\_STATUS** as 1'b1, per *Section 6.6.3.1*), or instruct the I3C Bus Controller Logic to stall the Bus (if possible) until the Driver can resolve the situation by freeing existing Data Chunks in the IBI Data Ring. If the Driver cannot resolve the situation in a timely manner, then the Host Controller shall terminate reception of the IBI payload. If a new IBI event occurs and the IBI Data Ring does not have any free Data Chunks, the Host Controller shall trigger the interrupt, and shall also NACK such an IBI event.

Within each Data Chunk, the Host Controller shall write the IBI payload data according to the current byte ordering mode that was selected, when the IBI was received and processed by the Host Controller’s I3C Bus Controller Logic, per *Section 6.8.3*.

Allocations in the IBI Data Ring are circular in nature, i.e., the oldest Data Chunk allocations are always freed first. The oldest allocation is associated with the oldest IBI Status Descriptor, at the position where the Dequeue Pointer for the IBI Status Ring is pointing. Field **CHUNK\_COUNTER** in register **CHUNK\_CONTROL** (see *Section 7.6.10.3*) shall provide the position for the first Data Chunk for this IBI Status Descriptor’s Data Chunk allocation. This register is read by the Driver, which shall allow the Host Controller to know when the Driver has started processing enqueued IBI Status Descriptor structures with Data Chunks (per *Section 6.9.2*).

The Driver shall also write to this register to notify the Host Controller that the Driver has consumed the oldest Data Chunk allocation, and intends to free the allocation that is associated with the oldest IBI Status Descriptor structure. The Host Controller shall clear this field (i.e., reset to zero) when the Ring Bundle is transitioned to the **Enabled** state.

Field **CHUNK\_COUNTER** allows the Host Controller and the Driver to communicate updates on freed Data Chunks, assuming that the Driver has read from register **CHUNK\_CONTROL** while the IBI Status Ring’s Dequeue Pointer was pointing to this IBI Status Descriptor structure (i.e., the structure that it intends to free), or some previous IBI Status Descriptor structure (per the conditions below for freeing adjacent IBI Status Descriptor structures):

- The Driver shall subsequently advance the Dequeue pointer, and shall write the new value to field **IBI\_DEQ\_PTR** in register **RH\_OPERATION1** (see *Section 7.6.10.10*), indicating that the Driver has consumed the oldest IBI Status Descriptor structure from the IBI Status Ring (per *Section 6.6.3.1*).
- The Driver shall then add the number of Data Chunks indicated by this IBI Status Descriptor (i.e., the value of field **CHUNKS**) to its cached value of register **CHUNK\_CONTROL**, and shall write this value back to register **CHUNK\_CONTROL**. This operation shall not use circular (i.e., “modulo”) arithmetic, as it would when advancing typical Ring pointers: the Driver simply needs to add the number of Data Chunks and write that new value back to the same register.

2124

**Note:**

2125     *The Driver may also free all Data Chunk allocations for several adjacent IBI Status Descriptor  
2126     structures, starting with the oldest, by advancing the Dequeue pointer by two or more positions and  
2127     writing the new value to field **IBI\_DEQ\_PTR** in register **RH\_OPERATION1**, and then updating the  
2128     Chunk Counter by adding a value equal to the total number of Data Chunks associated with all  
2129     such IBI Status Descriptor structures. See [Section 6.9.2](#).*

2130     *After writing to register **CHUNK\_CONTROL** to free the Data Chunk allocations for one or more  
2131     adjacent IBI Status Descriptor structures, the Driver shall read register **CHUNK\_CONTROL** again  
2132     before processing and freeing Data Chunks for the next IBI Status Descriptor structure, in order to  
2133     capture the location of the first Data Chunk and inform the Host Controller of the state at the time of  
2134     reading.*

2135     Field **CHUNK\_COUNTER** acts a monotonic counter. It may accept writes of values at least two times the  
2136     largest possible configured value of **CHUNK\_COUNT** in register **IBI\_SETUP** (see [Section 7.6.10.2](#)), i.e., values  
2137     larger than 2048. This ensures that the Driver can always perform the following operations:

- 2138     • Read the value of field **CHUNK\_COUNTER** at its largest possible value (i.e., position 0x3FE in the IBI Data  
2139       Ring).
- 2140     • Add the number of Data Chunks allocated for one or more IBI Status Descriptors, with each having up to  
2141       the largest possible Data Chunk allocation size (i.e., 0xFF, as indicated by an IBI Status Descriptor), and  
2142       adding up to the total number of Data Chunks in the IBI Data Ring, for a total of up to 0x3FE.
- 2143     • Write that new value (up to 0x7FC maximum) back to field **CHUNK\_COUNTER**, to indicate that up to the  
2144       maximum number of Data Chunks should be freed.

2145     A write to register **CHUNK\_CONTROL** notifies the Host Controller that the Driver has consumed and  
2146     subsequently freed one or more Data Chunk allocations for one or more IBI Status Descriptors, starting  
2147     with the IBI Status Descriptor structure that was pointed to by the Dequeue Pointer when the Driver read  
2148     register **CHUNK\_CONTROL**. The Host Controller does not need to inspect the contents of each such IBI  
2149     Status Descriptor structure (i.e., when the Driver advances the Dequeue Pointer) to determine how many  
2150     Data Chunks were originally allocated. Instead, the Host Controller shall maintain an internal “Chunk  
2151     Counter (HW)” pointer for the IBI Data Ring, and shall advance this pointer to free such Data Chunk  
2152     allocations, as the Driver writes an updated value to register **CHUNK\_CONTROL**.

2153

**Note:**

2154     *Implementation details of the internal “Chunk Counter (HW)” pointer are vendor-specific, and are  
2155     not covered by the scope of this I3C HCI Specification.*

2156     *Unlike the IBI Status Ring, the IBI Data Ring does not have an “enqueue” pointer that is visible to  
2157     the Host (i.e., readable by the Driver). The Host Controller shall maintain its own internal pointer for  
2158     the first available Data Chunk on the IBI Data Ring, and shall also ensure that this internal pointer  
2159     never goes past the Chunk Counter (either its internal pointer, or the value that might be read from  
2160     register **CHUNK\_CONTROL**).*

### 6.6.3.3 IBI Data Ring Configuration Changes

If the Driver wishes to change the IBI Ring data buffer chunk size or data buffer area, then the Driver shall perform the following steps:

1. Stop the Ring Bundle
2. Store the current position (i.e., value) of the Enqueue Pointer in Old Enqueue Pointer
3. Store the current data buffer chunk size in Old Data Chunk
4. Disable the Ring Bundle
5. Allocate the new data buffer, then update the Chunk Size and Data Head Pointer for the newly allocated buffer
6. Start/run the IBI Ring
7. Process the old IBI Ring until reaching Old Enqueue Pointer (still using Old Data Chunk for data chunk sizes)
8. Switch to the new Data Head and new Chunk Size
9. Enable the Ring Bundle
10. Free memory for the old data buffer area
11. Continue with the new buffer

Making the Data Chunk size variable allows for optimal memory use (i.e., set Chunk Size to the typical IBI payload size), simpler data buffer management, and very large IBI payloads (i.e., with a very large Data Chunk size and using all 255 IBI Status Descriptor entries, the maximum data buffer size can be very large, i.e., up to 261,888 bytes, with 256 bytes per Data Chunk and 1023 maximum Data Chunks, per [Section 7.6.10.2](#)).

The Data Chunk size can be regarded as a scaling factor: the same IBI Status Descriptor format can address either small or large data buffers. For smaller IBI payloads (which are generally expected), DWORD could be considered an optimal Data Chunk size.

The Driver should only extend the IBI Data Ring or make similar changes to the configuration under special circumstances, namely: (a) upon receiving a Hot-Join event, and (b) in systems where memory use optimization is critical.

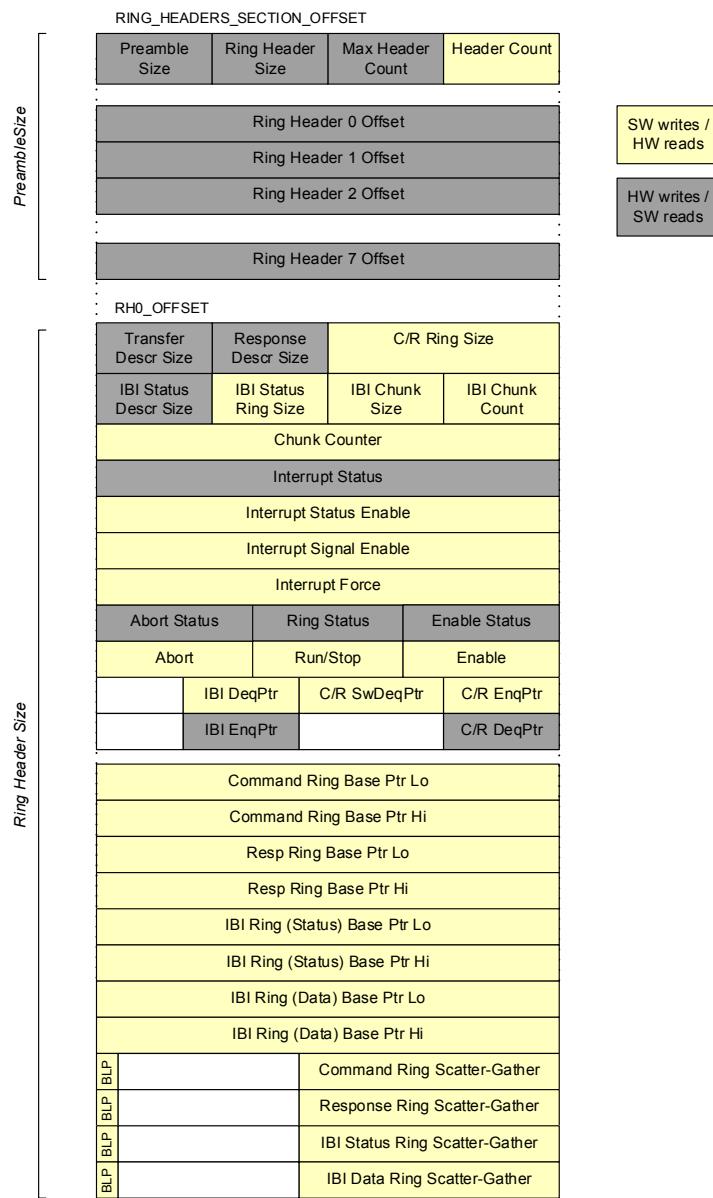
#### 6.6.4 Ring Header Registers

The Ring Headers section of the Host Controller memory map (at an offset defined in the Capabilities Registers section, see [Section 7.4.13](#)) begins with a Ring Headers Preamble register ([RHS\\_CONTROL](#), see [Section 7.6.1](#)) which contains fields for the number of Ring Headers supported ([MAX\\_HEADER\\_COUNT\\_CAPABILITY](#)) and the Ring Header Structure Size ([HEADER\\_SIZE](#)). The Preamble register also contains control field [MAX\\_HEADER\\_COUNT](#) to enable or disable the supported Ring Headers for use by software.

Following the Preamble register, the section contains offsets for up to eight Ring Headers in registers [RH0\\_OFFSET](#) through [RH7\\_OFFSET](#) ([Section 7.6.2](#) through [Section 7.6.9](#)).

Interrupt handling, memory pointers, pointer management, and Ring Bundle management registers are defined for each Ring Bundle in the Ring Header (see [Section 7.6.10](#)).

Please refer to [Section 7.6](#) for a detailed description of the Ring Headers Specific Registers.



2198

**Figure 20 Ring Header Registers**

## 6.6.5 Ring Bundle Servicing

If the Host Controller supports multiple Ring Bundles that are exposed as separate instances of Ring Header register sections (see *Section 6.6.4*), then the Ring Controller shall use arbitration logic to automatically manage the servicing of all Ring Bundles that are enabled for software (via register **RHS\_CONTROL**, see *Section 7.6.1*), configured for use with the Ring Header Base Address registers, and in an enabled and running state (via register **RH\_CONTROL**, see *Section 7.6.10.9*). However, this arbitration logic shall never select any Ring Bundles that are in a **DISABLED**, **STOPPED** or **ABORTED** state.

The Ring Controller's arbitration logic shall attempt to fairly balance the servicing of such enabled and running Ring Bundles, to ensure that all enqueued Transfer Descriptors are processed without undue delay. Additionally, the arbitration logic should avoid breaking up any enqueued command sequences (as defined in *Section 6.12*) if the last Transfer Descriptor for such a command sequence (i.e., a Command Descriptor with **TOC=1** that follows a Command Descriptor with **TOC=0**) was known to be enqueued by the arbitration logic.

However, for specific use cases, an implementer may define other prioritization algorithms or balancing controls, with appropriate registers in an implementer-defined Extended Capability structure. In general, the arbitration logic should balance the priority of the enabled and running Ring Bundles, while also not interrupting any command sequences in progress, breaking the atomicity of any single sets of Transfer Descriptors enqueued with a single "doorbell" event (unless otherwise allowed with other directives) or causing any undue interruption for the various transfers that comprise an I3C content protocol.

### 6.6.5.1 Ring Bundle Switching

Before switching from one valid (i.e., enabled and running) Ring Bundle to another valid Ring Bundle that has pending Transfer Descriptors yet to be processed, the Ring Controller's arbitration logic shall test to see whether the previously processed Command Descriptor did not drive a STOP condition or the HDR Exit Pattern (i.e., when **TOC=0**) and the indicated I3C Mode (i.e., field **MODE** in the Command Descriptor) is the same as for the previously processed Command Descriptor. In such cases, where the framing for the previous I3C Mode has not been ended and the Speed and/or Mode changes on switching Ring Bundles, the rules in *Section 6.2.5* of the I3C TCRI Specification [**MIP106J**] shall apply.

For cases when a Speed and/or Mode change requires the Host Controller to insert either a STOP condition or the HDR Exit Pattern in order to process the first enqueued Transfer Descriptor in the new Ring Bundle (i.e., due to a different value of field **MODE** in the Command Descriptor), or when any interrupts must be reported due to any other unexpected conditions that would have been caused by switching to another valid Ring Bundle, the Host Controller shall report such interrupts via the Ring Interrupt Status register in the Ring Header for the new Ring Bundle that the Ring Controller's arbitration logic had just switched to be processed.

### 6.6.5.2 Ring Bundle Locking

Software may temporarily lock the Ring Controller to remain on a single valid (i.e., enabled and running) Ring Bundle using the Ring Bundle Lock sub-command, sent as an Internal Control Command (see **Section 8.4.2.1** and **Table 136**) via the Command Ring of the Ring Bundle that will be locked.

- If another Ring Bundle is currently locked, then the Ring Controller shall reject the Ring Bundle Lock - sub-command, and return a Response Descriptor with error code 0x5 (**NACK**) in field **ERR\_STATUS**.
- If no Ring Bundle is currently locked, then the Host Controller shall lock to the current Ring Bundle and shall not switch to other enabled and running Ring Bundles:
  - The Ring Bundle ID to be locked shall be implicit, based on the Command Ring to which the software enqueues the Transfer Descriptor containing such an Internal Control Command.
  - If the Ring Bundle is not the current Ring Bundle that is being processed when software enqueues the Transfer Descriptor containing the Internal Control Command, then the Ring Controller shall attempt to lock to that Ring Bundle once the Ring Controller's arbitration logic switches to process it (per **Section 6.6.5.1**).
  - Once locked, the Ring Bundle shall remain locked until software enqueues a corresponding Ring Bundle Unlock sub-command, sent as an Internal Control Command via the same Command Ring of the currently locked Ring Bundle.
  - On receiving and successfully processing Ring Bundle Lock/Unlock sub-commands, the Ring Controller shall return a Response Descriptor with value 0x0 (**SUCCESS**) in field **ERR\_STATUS**.

**Note:**

*Once the Host Controller processes the Ring Bundle Lock sub-command (i.e., when that Command Ring is active and selected for processing by the Ring Controller's arbitration logic), the Host Controller shall not reject the Ring Bundle Lock sub-command, unless there is an internal error or some other configuration issue that prevents a successful locked state. In the rare event that this occurs, the Host Controller shall reject the Ring Bundle Lock sub-command and return a Response Descriptor with error code 0x5 (**NACK**) in field **ERR\_STATUS**.*

*By definition, the Host Controller shall not process the Command Rings for any other valid Ring Bundles, so as a result there can be no contention for the locked state, and also no rejection of a Ring Bundle Lock sub-command due to another Ring Bundle currently being locked.*

While any Ring Bundle is operating in a locked state, if a Ring Bundle encounters an error and is halted, stopped or otherwise interrupted; or if the Ring operations are aborted by software (i.e., by writing to field **ABORT** in register **RH\_CONTROL**, see **Section 7.6.10.9**), the Ring Controller's arbitration logic shall not switch to any other Ring Bundles that might be enabled and in a running state and have pending Transfer Descriptors (i.e., those that might otherwise be available for Command/Response processing) until and unless the locked Ring Bundle is unlocked, or operation is resumed.

Each Ring Header for a valid Ring Bundle shall indicate its current Lock/Unlock status in field **LOCKED** in register **RH\_STATUS** (see **Section 7.6.10.8**).

### 6.6.6 Ring Abort Operation

The Driver may trigger the Ring Abort operation for a Ring Bundle by setting the **ABORT** bit to 1'b1 in register **RH\_CONTROL** (see *Section 7.6.10.9*) for the associated Ring Header.

If the Ring Bundle is valid (i.e., enabled and running) and is currently the active Command/Response Ring for processing, then:

- The Host Controller shall abort (i.e., shall terminate) any in-progress transfers for this Ring Bundle that the I3C Bus Controller Logic is currently driving, at the earliest opportunity.

The I3C Bus Controller Logic shall terminate transfers according to the requirements of the I3C Specification for the current I3C Mode (i.e., without causing any I3C Bus errors).

The transactions that were aborted shall be indicated accordingly in the corresponding Response Descriptors, with field **ERR\_STATUS** having a value of 0x8 (**HC\_ABORTED**). The Host Controller shall also trigger the **TRANSFER\_ABORT\_STAT** interrupt (if enabled for notification) for the associated Ring Header, via register **RH\_INTR\_STATUS** (see *Section 7.6.10.4*).

- The Host Controller shall also indicate that the Ring Bundle's state has changed to **ABORTED**, by asserting the **RING\_OP\_STAT** interrupt (if enabled for notification) for the associated Ring Header (see *Section 7.6.10.4*).

**Note:**

*If the Host Controller is instructed to cancel a transaction sequence (i.e., Transfer Descriptor contains a Command Descriptor with field TOC=0) in order to process the Ring Abort operation, then it shall also trigger an interrupt via field HC\_SEQ\_CANCEL\_STAT in register INTR\_STATUS (see Section 7.4.7) to inform the Host that the sequence was terminated before processing a Transfer Descriptor containing a Command Descriptor having field TOC=1 (see Section 6.12).*

If the Ring Bundle is valid (i.e., is enabled and running) but is not currently active for processing; or if the Ring Bundle was enabled but not currently running, then:

- No transactions shall be aborted, since no Transfer Descriptor was being actively processed by the I3C Bus Controller Logic.
- The Host Controller shall indicate that the Ring Bundle's state has changed to **ABORTED**, by asserting the **RING\_OP\_STAT** interrupt (if enabled for notification) for the associated Ring Header. The Ring Bundle shall immediately be removed from the list of valid Ring Bundles that may be selected by the Ring Controller's arbitration logic for servicing (per *Section 6.6.5*).

Once a Ring Bundle receives the Ring Abort request and stops processing Transfer Descriptors as a result (if it was active for processing), it shall trigger its **RING\_OP\_STAT** interrupt to indicate its **ABORTED** state, as indicated above. The Driver shall perform any needed Ring recovery steps upon receiving this notification.

**Note:**

*Any subsequent Transfer Descriptors that might have been enqueued after any aborted transfers shall remain unprocessed in the Command Ring once the Ring Bundle enters the ABORTED state. The Driver should determine whether such Transfer Descriptors should remain enqueued for future processing, or replaced with other Transfer Descriptors (or, in some situations, replaced with "NoOp"-type Internal Control Commands; see Section 8.4.2).*

2305 To resume operation for the Ring Bundle, the Driver shall perform the following steps:

- 2306 1. Clear the Ring Operation Status interrupt notification for the Ring Header, by writing 1'b1 to field  
2307 **RH\_OP\_STAT** in register **RH\_INTR\_STATUS** (*Section 7.6.10.4*).  
2308 2. Clear the Transfer Abort Status interrupt notification for the Ring Header, if any transfers were aborted  
2309 (as specified above), by writing 1'b1 to field **TRANSFER\_ABORT\_STAT** in register **RH\_INTR\_STATUS**  
2310 (*Section 7.6.10.4*).  
2311 3. Clear the Ring Header's Ring Abort Request state, by writing 1'b0 to field **ABORT** in register  
2312 **RH\_CONTROL** (*Section 7.6.10.9*).  
2313 4. Write to the Ring Header's Enqueue Pointer for the Command/Response Ring, i.e., field **CR\_ENQ\_PTR**  
2314 in register **RH\_OPERATION1** (*Section 7.6.10.10*). This automatically resumes the Ring operation, and  
2315 any remaining Transfer Descriptors that are enqueued will be processed, per *Section 6.8.2.*, unless the  
2316 Ring Bundle entered its **ABORTED** state via the Host Controller Abort operation (see *Section 6.8.4*) or  
2317 the Host Controller remains in **ABORTED** state.

2318 **Note:**

2319 A Ring Abort can also be triggered by a global Host Controller Abort operation (see *Section 6.8.4*).  
2320 If this occurs, then both register **HC\_CONTROL** (at the global level) and register **RH\_CONTROL** (for  
2321 this Ring Bundle) shall report the **ABORTED** state. However, if only a Ring Abort was triggered on its  
2322 own (i.e., via register **RH\_CONTROL**) then this does not necessarily cause, nor necessarily imply, a  
2323 global Host Controller Abort operation.

## 6.7 Transfer Command Handling

A Host Controller includes I3C Bus Controller Logic and allows the Host to enqueue transfers via its Transfer Command/Response Interface (TCRI). While the Host Controller is the Active Controller of the I3C Bus, these transfers are driven to the I3C Bus by the Controller Logic, and the response status for each transfer can be read by the Host to determine whether the transfer succeeded or failed. This mode of operation is called Active Controller mode.

For each enqueued transfer, the Controller Logic uses the transfer parameters that are encoded in the Command Descriptor structure. These parameters inform the Controller Logic which I3C/I<sup>2</sup>C Target Device is the recipient of the transfer, or (for Broadcast CCC commands) inform the Controller Logic that a transfer is directed to all I3C Target Devices on the I3C Bus.

**Note:**

*Additional details on Transfer Command and Transfer Response handling are defined in Section 6.2 of the I3C TCRI Specification [MIPI06].*

In DMA Mode, since Command/Response Rings are bundled together into Ring Pairs, for every Command Descriptor structure populated on the Command Ring there shall be exactly one Response Descriptor structure populated on the corresponding Response Ring. The Driver provides the Transfer Descriptor in Driver-allocated memory. Refer to **Section 6.6** for more details.

In PIO Mode, the Command Descriptor structure is written to registers (the Command Queue Port), and related data is read from/written to the Transfer Data Port. Refer to **Section 6.7** for more details.

The size of the Command Descriptor is 2 DWORDs, and the size of the Response Descriptor is 1 DWORD. The format of the Command Descriptor is “Format 1: Legacy Format, Indexed” as defined in **Section 7.1** of the I3C TCRI Specification [MIPI06].

**Note:**

*It is possible that future versions of this I3C HCI Specification might use different formats and/or sizes for the Command Descriptor. In that event, the new format(s) and/or size(s) will be indicated by a different value in field **CMD\_SIZE**.*

The Command Descriptor is defined in **Section 8.4**, and supports several types of Transfer Commands per **Section 7.1.2** of the I3C TCRI Specification [MIPI06], in addition to the Internal Control type command, as indicated by field **CMD\_ATTR**. Transfers are limited to 64 KB in size. For all Transfer Commands, the Command Descriptor includes a 5-bit **DEV\_INDEX** field containing an index into the DAT table:

- **For private transfers and Direct CCCs:** The Host Controller shall fetch the Dynamic Address from the DAT entry indicated by the **DEV\_INDEX** field. Software shall provide a valid index to a DAT entry that has been populated with a valid Dynamic Address of a Device on the I3C Bus (or an assigned Group Address for Write-type transfers, if supported).
- **For Broadcast CCCs:** The Host Controller shall not use the **DEV\_INDEX** field. Software should set **DEV\_INDEX** to 5'h00.

**Note:**

*This version of the I3C HCI Specification does not support transfers using some of the new features enabled by version 1.1.1+ of the I3C Specification [MIPI05], such as HDR-BT (Bulk Transfer) Mode, Multi-Lane transfers in any supported I3C Modes, CCC flows in HDR Modes, or several types of Combo transfers that are necessary for specific use cases such as Device to Device Tunneling. Such transfer types or control capabilities may be added as extended capabilities, or could be supported in a future version of this I3C HCI Specification.*

## 6.8 Transfers

### 6.8.1 Transfers in PIO Mode

In PIO Mode, the Driver shall enqueue a Command Descriptor structure by writing it to the Command Queue Port (i.e., register **COMMAND\_QUEUE\_PORT**, see *Section 7.5.1*), starting with its least significant DWORD (since the Command Queue Port register is 32 bits wide). Before enqueueing the Command Descriptor structure, the Driver shall put the corresponding write data into the Tx Data Queue via the Transfer Data Port (i.e., register **XFER\_DATA\_PORT**, see *Section 7.5.3*), for Transfer Commands that are a Write-Type transfer request and also require the Tx Data Queue. After the transfer is completed, the Driver shall read its status from the Response Queue Port (i.e., register **RESPONSE\_QUEUE\_PORT**, see *Section 7.5.2*) as notified by a **RESP\_READY\_STAT** interrupt. Additionally, if the Transfer Command is a Read-Type transfer request, the Driver shall read any incoming data from the Rx Data Queue via the Transfer Data Port.

The Tx Data Queue is drained by the Host Controller, whereas the Rx Data Queue is drained by Driver. Since the Tx/Rx Data Queues are finite in size, the Driver shall ensure that the I3C Bus Controller Logic does not encounter any situations where a transaction would be aborted due to Data Queue underflow (for Write-Type transfers) or overflow (for Read-Type transfers).

Before initiating a Write-Type transfer, the Driver shall ensure that there is data in the Tx Data Queue for the hardware to read: either by limiting the request size and putting all data into the queue when possible, or else by initially loading the Tx Data Queue with partial data, and subsequently populating the Tx Data Queue with remaining data as the I3C Bus Controller Logic begins transferring the data on the I3C Bus, to avoid an underflow during the Write-Type transfer.

For Write-Type transfers that are larger than the Tx Data Queue size, the Driver should set the Transmit Buffer threshold (i.e., field **TX\_BUF\_THLD** in register **DATA\_BUFFER\_THLD\_CTRL**; see *Section 7.5.6*) in order to remain responsive and prevent underflow conditions. The Driver then populates the Tx Data Queue up to its full size (i.e., field **TX\_DATA\_BUFFER\_SIZE** in register **QUEUE\_SIZE**, see *Section 7.5.7*), enqueues the Command Descriptor, and then waits for the Tx FIFO to drain. When the Driver receives the **TX\_THLD\_STAT** interrupt, it knows that the Host Controller can accept additional data, so it writes more data into the TX Data Queue, up to the threshold value. In order to prevent an underflow condition and the resulting early transaction termination, for transactions larger than the size of the Tx Data Queue, the Driver shall ensure that the Tx Data Queue does not drain completely before it writes the last DWORD of data for the transfer.

**Note:**

Per *Section 6.6.5*, the Host Controller shall use the **TX\_THLD\_STAT** interrupt to indicate its readiness to accept additional data DWORDs in the Tx Data Queue, up to the threshold level provided in **TX\_BUF\_THLD**. When the Host receives this interrupt, it knows that it may write up to that number of DWORDs of data into the Tx Data Queue.

Similarly, for Read-Type transfers, the Driver should set the Receive Buffer threshold (i.e., field **RX\_BUF\_THLD** in register **DATA\_BUFFER\_THLD\_CTRL**, see *Section 7.5.6*) in order to remain responsive and prevent overflow conditions. The Driver then enqueues the Command Descriptor and waits for an interrupt notification that indicates that incoming data is ready to be read from the Rx Data Queue (i.e., field **RX\_THLD\_STAT** in register **PIO\_INTR\_STATUS**, see *Section 7.5.9*). The frequency of such interrupt notifications is determined by the value written into field **RX\_BUF\_THLD** (i.e., the Receive Buffer threshold). After receiving this interrupt notification, the Driver then reads the incoming data from the Rx Data Queue. In order to prevent an overflow condition and the resulting early transaction termination, the Driver shall ensure that incoming data is read from the Rx Data Queue before it becomes totally filled with data (i.e., field **RX\_DATA\_BUFFER\_SIZE** in register **QUEUE\_SIZE**, see *Section 7.5.7*).

2410 **Note:**

2411 Per **Section 6.6.5**, the Host Controller shall use the **RX\_THLD\_STAT** interrupt to indicate that some  
2412 data DWORDs (i.e., at least the number indicated in the threshold level provided in **RX\_BUF\_THLD**)  
2413 are in the Rx Data Queue. When the Host receives this interrupt, it knows that it may read up to  
2414 that number of data DWORDs from the Rx Data Queue.

2415 However, under certain circumstances, the last data DWORD for a particular Read-Type transfer  
2416 might not trigger the **RX\_THLD\_STAT** interrupt, since the minimum threshold level that may be  
2417 configured for **RX\_BUF\_THLD** for such interrupt notifications is 2 DWORD entries in the Rx Data  
2418 Queue (see **Section 7.5.6, Table 42**). This is especially pertinent for very short Read transfers (i.e.,  
2419 where **DATA\_LENGTH** ≤ 4 bytes), as such data DWORDs might never reach the threshold and  
2420 therefore not trigger the **RX\_THLD\_STAT** interrupt notification.

2421 The Driver shall inspect the Response Descriptor, check field **DATA\_LENGTH** to determine the  
2422 actual number of data bytes received, and consume an appropriate number of DWORDs from the  
2423 Rx Data Queue for each Read-Type transfer that has completed, even if not all such DWORDS  
2424 would have generated the **RX\_THLD\_STAT** interrupt. The Driver must also appropriately configure  
2425 the Response Ready Buffer Threshold (i.e., field **RESP\_BUF\_THLD** in register **QUEUE\_THLD\_CTRL**,  
2426 see **Section 7.5.5, Table 45**) to ensure that it receives **RESP\_READY\_STAT** interrupt notifications in  
2427 a timely manner.

2428 Interrupts related to PIO Mode transfers are defined in the PIO Registers Section (see **Section 7.5**). Note  
2429 that the PIO interrupt for Rx Data Buffer Threshold Status might be asserted before a Read-Type transfer is  
2430 complete, for larger transfers, depending on the threshold settings in register **QUEUE\_THLD\_CTRL** (see  
2431 **Section 7.5.5**). The Driver should be ready to read such incoming data in advance of receiving the  
2432 Response Descriptor that describes this data, including its overall message length.

2433 Upon completion of the transfer, the Host Controller might indicate that a Response Descriptor is ready to  
2434 be read, by setting field **RESP\_READY\_STAT** in register **PIO\_INTR\_STATUS** (see **Section 7.5.9**). This field  
2435 might be automatically cleared upon the Driver reading out one or more Response Descriptors from the  
2436 Response Queue (i.e., the readout operation might serve as an interrupt clear). Response Descriptor  
2437 interrupts shall depend on the threshold setting for the Response Queue (i.e., field **RESP\_BUF\_THLD** in  
2438 register **QUEUE\_THLD\_CTRL**).

2439 The Driver shall correlate the data DWORDs read from the Rx Data Queue to ensure that the data is  
2440 correctly associated with the read transfer for the length indicated by the Response Descriptor. For I3C  
2441 Read-type transfers, a successful Response Descriptor (i.e., field **ERR\_STATUS** having a value of 0x0)  
2442 might indicate a ‘short’ Read-Type transfer, per **Section 6.2.7** of the I3C TCRI Specification [**MIPI06**]. In  
2443 PIO Mode, the length of data received from the I3C Target Device would typically be fully understood in  
2444 context only after most (or, in some cases, all) of the data DWORDs were been read from the Rx Data  
2445 Queue and then compared with the Response Descriptor. Since the relevant PIO Queues are independently  
2446 read for Read-Type transfers, but have an implied association of data that is available to be consumed, the  
2447 Driver might not always read from the Rx Data Queue and the Response Queue in the same order, due to  
2448 the threshold settings for such queues, as well as the presence of other enqueued Transfer Commands.  
2449 However, the Driver must ensure that it correctly associates the received data DWORDs with the Response  
2450 Descriptor for each Read-Type transfer, in order to determine whether it has received all the expected data  
2451 DWORDs, or whether this is a ‘short’ Read-Type transfer.

2452 The Host Controller shall also generate interrupts for other Queue events, which shall be set and cleared  
2453 automatically based on operations involving these Queues, depending on the threshold settings in register  
2454 **QUEUE\_THLD\_CTRL** (see **Section 7.5.5**). The Host Controller shall also generate interrupts for other PIO  
2455 interrupt events that must be explicitly cleared by the Driver:

- 2456 • Interrupt **TRANSFER\_ABORT\_STAT**: Transfer was aborted  
2457 • Interrupt **TRANSFER\_ERR\_STAT**: Error

For transfers that are aborted by the I3C Bus Controller Logic due to either an underflow condition in the Tx Data Queue, or an overflow condition in the Rx Data Queue (as described above), the Host Controller shall stop processing any subsequent Command Descriptors in the Command Queue after reporting this error. Additionally, if the aborted transfer was not the last transfer in a sequence (i.e., field **TOC**=0, as defined in [Section 6.12](#)) then the Host Controller shall also report this as a cancelled transaction sequence, per [Section 6.12.1](#).

### 6.8.2 Transfers in DMA Mode

**Note:**

*This section applies only for Host Controller implementations that implement DMA Mode, and thus connect to a System Bus that supports memory-mapped IO and DMA.*

In DMA Mode, the transfer to a Device is performed via Command and Response Rings, per [Section 6.6.2](#). The Driver shall populate the Command Ring with Transfer Descriptor structures, and eventually update the Enqueue Pointer for the related Ring Bundle (see [Table 4](#)).

The Transfer Descriptor supports either:

- A Data Buffer Pointer to a single, physically contiguous memory block that data is fetched from and/or stored to, or
- A pointer to a Driver-allocated table of Data Buffer Pointers containing the Addresses of multiple, potentially non-contiguous memory blocks (i.e., a Scatter-Gather List).
  - The Data Buffer Pointers shall take the form of an array of Memory Descriptor structures (see [Section 8.7](#)). This array shall be a single, physically contiguous allocation of memory.
  - Each Memory Descriptor shall contain an Address of a memory buffer, that the Driver shall allocate. Each memory buffer shall be DWORD-aligned.

If a Transfer Descriptor uses a Scatter-Gather List, then there are two limitations on the sizes of the set of memory buffers that the Driver shall allocate for the Scatter-Gather structures:

- The total size of the Data Buffer Pointers table shall be a multiple of the Memory Descriptor structure size; and
- The sum of the memory buffer sizes in a Scatter-Gather List shall be  $\leq$  the maximum transfer size.

Additionally, each Data Buffer Pointers table should have a limited number of entries, i.e., to avoid memory fragmentation where possible. The Data Buffer Pointers table should be  $\leq$  0xFF entries, in order to limit the processing time for each Transfer Descriptor and reduce the number of memory read operations required to fetch all such Memory Descriptors. Although the maximum length is 0xFFFF (i.e., field **BLOCK\_SIZE** in the Transfer Descriptor, when field **BLP** = 1'b1), the Driver should constrain this value to 0xFF or lower (see [Figure 49](#)).

An update of the Enqueue Pointer shall serve as a Doorbell for the Host Controller, which shall then evaluate the number of Transfer Descriptor entries to process. For each Transfer Descriptor that has been enqueued, the Host Controller shall perform the described transaction on the Bus. The order and sequence of such Transfer Descriptors shall be maintained per [Section 6.12](#). For transactions with a data phase, the Host Controller shall move the data based on the Data Buffer (i.e., either as a single region, or Scatter-Gather, as indicated in the Transfer Descriptor). The Driver shall set field **WROC** in the Command Descriptor structure to 1'b1 for every command scheduled in the Command Ring.

**Example:** Assume that the Command/Response Rings are initialized, with both the Enqueue Pointer and Dequeue Pointer having value 0. The Driver advances the Enqueue Pointer to value X, which triggers a Doorbell notification.

- First the hardware processes all Transfer Descriptor entries, in order, from 0 through X-1 (i.e., X Transfer Descriptors are processed).
- Next, the Driver advances the Enqueue Pointer by Y, to X+Y, to schedule Y additional commands.
- After this write, the hardware processes Transfer Descriptor entries from X through X+Y-1 (i.e., Y additional Transfer Descriptors are processed).

For every Transfer Descriptor on the Command Ring, the Host Controller shall populate one Response Descriptor on the Response Ring, indicating the status of the corresponding Bus transfer. When requested in the Transfer Descriptor (by setting the **IOC** bit to 1'b1), the Host Controller shall issue an interrupt to notify the Driver that outstanding Response Descriptors are available and should be processed. The Host Controller may process any number of outstanding Command Descriptor structures, in order, until reaching the Enqueue Pointer (and no further). The number of processed Command Descriptor structures is reflected in the Dequeue Pointer.

The Driver shall notify the Host Controller of the number of Responses processed by advancing the Software Dequeue Pointer. The Driver may process an arbitrary number of Responses, in order, until reaching the Dequeue Pointer (and no further). This approach allows the Host Controller to re-generate the interrupt when necessary.

**Example:** Assume that the hardware has produced  $N$  Response Descriptors, and that the software then receives the interrupt to process them, but for some reason the Driver is only able to process  $M$  of the Response Descriptors (where  $M < N$ ). When the Host Controller detects that the Driver has advanced the Software Dequeue Pointer, it will know that  $N - M$  additional Responses are still pending for processing in Response Ring, and as a result the Host Controller may generate a new interrupt to remind the Driver to continue processing the remaining  $N - M$  Response Descriptors.

When the Driver transitions a Ring Header to the **Enabled** state, the Host Controller shall set its own Dequeue Pointer to 0 and then wait for a write to the Enqueue Pointer. Upon transition to the **Enabled** state, the Host Controller shall re-read the Base Pointers for all Rings in that Ring Bundle.

A Command/Response Ring Pair is considered idle when the Enqueue Pointer, Dequeue Pointer, and Software Dequeue Pointer are all equal to one another.

To prevent Commands on the Ring from ever overflowing when advancing the Enqueue Pointer, the Host Controller Driver shall never permit the Enqueue Pointer to reach the value of the Software Dequeue Pointer. However, advancing the Software Dequeue Pointer may result in the Software Dequeue Pointer having the same value as the Enqueue Pointer.

The Host Controller processes Ring Bundles individually:

- The number of enabled Ring Bundles is set in the Ring Headers Preamble registers (*Section 7.6.1*). E.g., the Driver shall configure the Host Controller to only look at enabled Ring Bundles: for example, 0, 1, 2, and 3.
- The order in which the Host Controller starts to process Ring Bundles, and the order in which the Host Controller processes subsequent Ring Bundles, shall be hardware-implementation specific.
- The Host Controller shall use its Ring Controller arbitration logic to switch processing to the next Ring Bundle that is valid (i.e., enabled and running) and has pending Transfer Descriptors to be processed, per *Section 6.6.5*.

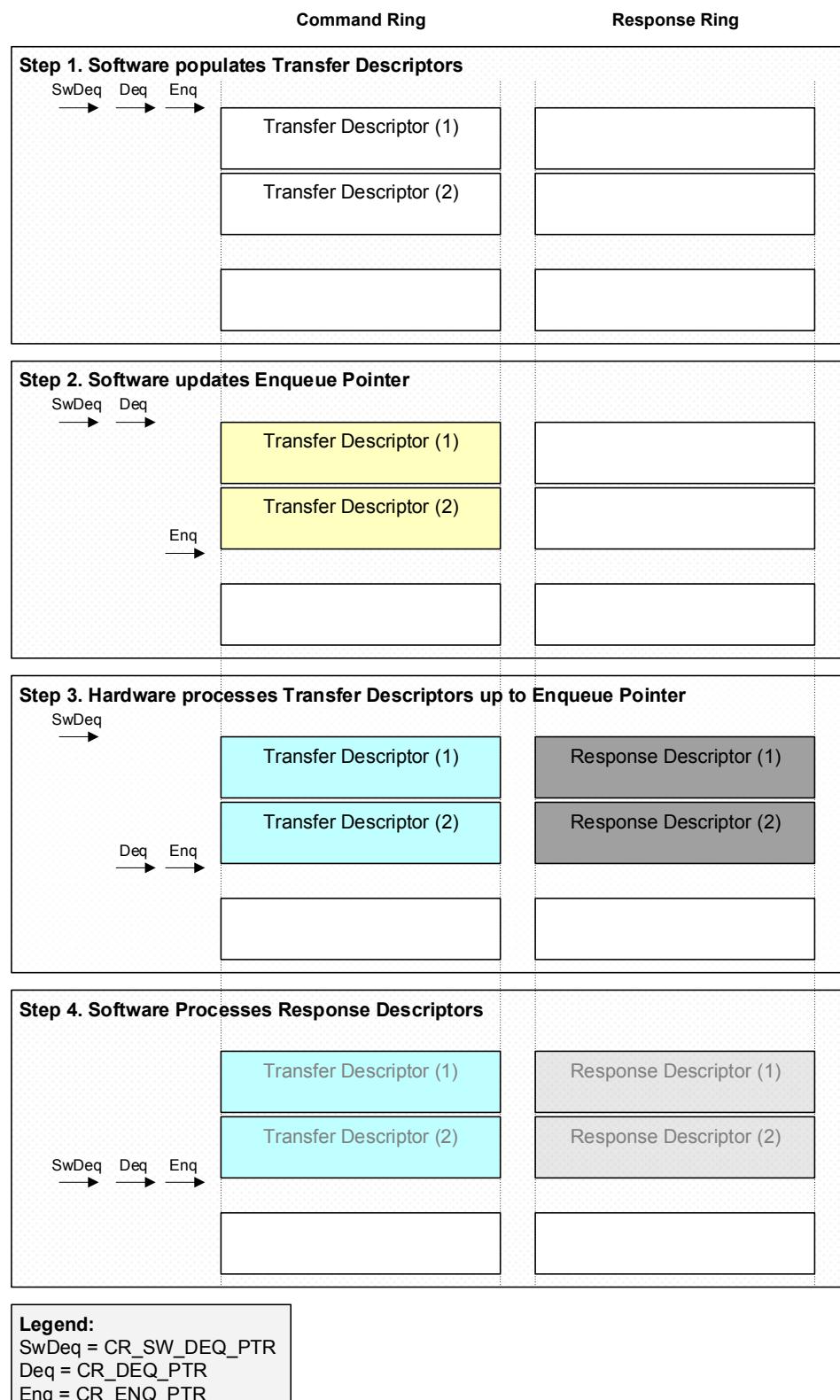
The Host Controller shall have only one Ring Headers registers section per Host Controller Interface, containing only one Preamble, with up to eight Ring Bundle offsets (as defined by the Ring Header Descriptor registers). All Ring-specific registers are defined in the respective Ring Headers. Driver-allocated memory is pointed to by the Ring Base Address High and Low registers for the Command, Response, IBI, and Data Rings (*Section 7.6.10.12* through *Section 7.6.10.19*).

Registers related to Ring Bundle interrupts are defined in the Ring Header. Completion of a transfer (i.e., one resulting from a Transfer Descriptor with the **IOC** bit set) triggers the **TRANSFER\_COMPLETION** interrupt.

The Driver may trigger the Ring Abort operation for a Ring Bundle by setting the **ABORT** bit to 1'b1 in register **RH\_CONTROL** (see *Section 7.6.10.9*). See *Section 6.6.6* for more details.

2550      **Figure 21** illustrates Ring and pointer updates during Command processing:

- 2551      • **Step 1:** The Ring is idle (i.e., the Enqueue Pointer, Dequeue Pointer, and Software Dequeue Pointer are  
2552      all equal).
- 2553      • **Step 2:** Software enqueues a new transaction by advancing the Enqueue Pointer.
- 2554      • **Step 3:** Upon completion of the Bus transaction, and availability of the Response status, the Host  
2555      Controller populates the corresponding Response Descriptor structure. Note that the position of this  
2556      Response Descriptor within the Response Ring is the same as the position of the Transfer Descriptor  
2557      within the Command Ring.
- 2558      • **Step 4:** When the Driver completes processing of pending Response Descriptors in the Response Ring, it  
2559      indicates its new position in Ring by advancing the Software Dequeue Pointer. At this point the Enqueue  
2560      Pointer, Dequeue Pointer, and Software Dequeue Pointer are again all equal, so the Ring returns to the  
2561      idle state.



2562

Figure 21 Transfer in DMA Mode

### 6.8.3 Data Byte Ordering

For Transfer Commands, the data being sent or received is specific to the application and the particular Target Device, and interpreted as an opaque stream of bytes by the Host Controller. The Host Controller interface is defined in terms of DWORD entries (i.e., 32-bit wide data words) that are used to store up to 4 bytes per entry.

The particular byte within a DWORD that is read or written first (i.e., the lowest offset in the DWORD entry) is system dependent, and implementers should determine the correct ordering to use for a particular system architecture and its system bus. Systems that use the Least Significant Byte first are commonly called Little Endian, and systems that use the Most Significant Byte first are commonly called Big Endian.

To make data transfers as efficient as possible, a Host Controller should accommodate the same byte ordering used by its connected Host system. Implementers should consider the integration options and offer byte ordering capabilities that automatically handle the byte ordering of data DWORDs in a manner best suited to the connected Host system.

Field **DATA\_BYTE\_ORDER\_MODE** in register **HC\_CONTROL** (see *Section 7.4.2*) indicates the supported byte ordering mode. The Host Controller shall use this byte ordering mode with the following uses:

- **For PIO Mode:**

- For Write-Type transfers, all Data DWORDs written by the Host into the Tx Data Queue (i.e., the Transfer Data Port, register **XFER\_DATA\_PORT**; see *Section 7.5.3*)
- For Read-Type transfers, all Data DWORDs read by the Host from the Rx Data Queue (i.e., the Transfer Data Port, register **XFER\_DATA\_PORT**; see *Section 7.5.3*)
- For IBIs, all IBI Data DWORDs read by the Host from the IBI Port (i.e., register **IBI\_PORT**; see *Section 7.5.4*)

**Note:**

*The IBI Status Descriptor shall not be affected or otherwise changed by the current byte ordering mode when read from the IBI Port, as this data structure is a discrete 32-bit entity.*

- **For DMA Mode:**

- For Write-Type transfers, all Data DWORDs read by the Memory Access Interface from Host system memory, at addresses indicated by the Transfer Descriptor (see *Section 8.3*)
- For Read-Type transfers, all Data DWORDs written by the Memory Access Interface to Host system memory, at addresses indicated by the Transfer Descriptor (see *Section 8.3*)
- For IBIs, all IBI Data DWORDs written by the Memory Access Interface to Host system memory, at addresses indicated by the IBI Data Ring's Base Address and the Host Controller's internal register of free chunks (see *Section 6.9.2*)

The Host Controller shall not apply the byte ordering mode to change the contents of any other registers, descriptors, or data structures that are read/written via mailbox registers (such as Queues in PIO Mode) or within other direct memory operations (such as Rings in DMA Mode). Such other registers, descriptors, and data structures shall be read or written in their native byte ordering, as discrete entities that are defined in terms of 32-bit DWORDs on all systems.

The order of data transmission of a data byte on the I3C Bus shall always follow the I3C Specification (i.e., SDR Mode proceeds from bit 7 to bit 0), regardless of the current byte ordering mode.

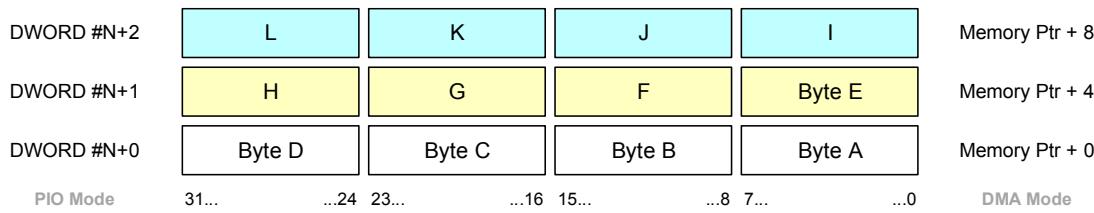
2602 **Little-Endian Byte Ordering Mode**

2603 For Write-Type transfers, the I3C Bus Controller Logic shall drive Data bytes from each DWORD entry in  
2604 the Tx Data Buffer, starting with the least significant byte (i.e., Bits[7:0]) and continuing on to the most  
2605 significant byte (i.e., Bits[31:24]).

2606 If the transfer length is not DWORD-aligned (i.e., is not an even multiple of 4 bytes) then the I3C Bus  
2607 Controller Logic might end the transfer before reaching the most significant byte within the last DWORD  
2608 entry, according to the framing and data transfer coding for the I3C Mode.

2609 For Read-Type transfers, the I3C Bus Controller Logic shall read Data bytes from the Target Device and  
2610 store them into the Rx Data Buffer, starting with the least significant byte and continuing on to the most  
2611 significant byte. If the read transfer ends before the DWORD boundary, then the remaining byte locations  
2612 in that DWORD entry shall not be used.

2613 **Figure 22** shows the order in which the Data bytes appear on the I3C Bus during a Transfer Command's  
2614 transaction.



2615 **Figure 22 Data Payload Byte Ordering (Little Endian)**

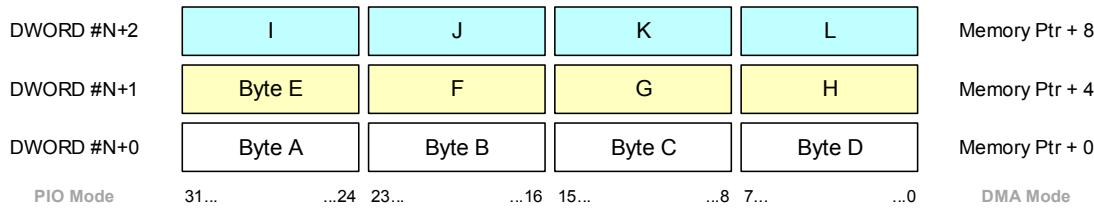
2617 **Big-Endian Byte Ordering Mode**

2618 For Write-Type transfers, the I3C Bus Controller Logic shall drive Data bytes from each DWORD entry in  
2619 the Tx Data Buffer, starting with the most significant byte (i.e., Bits[31:24]) and continuing on to the least  
2620 significant byte (i.e., Bits[7:0]).

2621 If the transfer length is not DWORD-aligned (i.e., is not an even multiple of 4 bytes) then the I3C Bus  
2622 Controller Logic might end the transfer before reaching the least significant byte within the last DWORD  
2623 entry, according to the framing and data transfer coding for the I3C Mode.

2624 For Read-Type transfers, the I3C Bus Controller Logic shall read Data bytes from the Target Device and  
2625 store them into the Rx Data Buffer, starting with the most significant byte and continuing on to the least  
2626 significant byte. If the read transfer ends before the DWORD boundary, the remaining byte locations in that  
2627 DWORD entry shall not be used.

2628 **Figure 23** shows the order in which the Data bytes appear on the I3C Bus during a Transfer Command's  
2629 transaction.



2630 **Figure 23 Data Payload Byte Ordering (Big Endian)**

#### 6.8.4 Host Controller Abort Operation

The Driver may initiate a Host Controller Abort operation at any time, while enqueued Command Descriptors are waiting to be processed and also while the I3C Bus Controller Logic is currently processing active transfers. For certain situations, the Driver might initiate the Host Controller Abort on detecting an incorrect result or other status received for a Read-Type transfer (i.e., a Private Read, a Direct Read CCC, or a Direct GET CCC) that would not halt operation automatically (i.e., a NACK error) or is not necessarily an I3C Bus error, but would have otherwise been determined to be an error or a sign of operations diverging from an expected result (i.e., from software).

The Driver may initiate the global Host Controller Abort operation by setting the **ABORT** bit to 1'b1 in register **HC\_CONTROL** (see *Section 7.4.2*). The Host Controller shall receive this request and process it according to the current operating mode. In effect, setting the **ABORT** bit acts as a method for cancelling the I3C Bus Controller Logic's current transfer, and also for preventing the I3C Bus Controller Logic from processing any other enqueued transfers in any Rings/Queues (per the operating mode), or for Scheduled Commands (if supported; see *Section 6.16*).

- **In any operating modes:**

- The Host Controller shall abort (i.e., shall terminate) any in-progress transfers that the I3C Bus Controller Logic is currently driving or receiving, at the earliest opportunity.

The I3C Bus Controller Logic shall terminate transfers according to the requirements of the I3C Specification for the current I3C Mode (i.e., without causing any I3C Bus errors).

The transactions that were aborted are indicated accordingly in the corresponding Response Descriptors, with field **ERR\_STATUS** having a value of 0x8 (**HC\_ABORTED**) or another transfer-type specific value; and the Host Controller shall also assert the **TRANSFER\_ABORT\_STAT** interrupt (if enabled for notification) for the current operating context(s) affected by the Host Controller Abort operation:

- **For PIO Mode** this is register **PIO\_INTR\_STATUS** (see *Section 7.5.9*).
- **For DMA Mode** this is register **RH\_INTR\_STATUS** (see *Section 7.6.10.4*) in the Ring Header for the Ring Bundle that was currently active for processing.

- Once received, a Host Controller Abort request cannot be cancelled by the Driver (i.e., before it is either processed or propagated, as defined below).
- The Host Controller shall enter the **ABORTED** state:

- Bit **ABORT** in register **HC\_CONTROL** shall read with a value of 1'b1 while the Host Controller remains in the **ABORTED** state.

- **Additionally, in PIO Mode:**

- If the Command Queue contains any subsequent Command Descriptors that might have been enqueued after the aborted transfers (i.e., those that were stopped, and indicated as aborted with a Response Descriptor typically having a value of 0x8 (**HC\_ABORTED**) or another transfer type specific value), then these subsequent Command Descriptors shall remain in the Command Queue. The Driver shall determine whether to leave such Command Descriptors intact, or whether to reset the Command Queue (i.e., by using register **RESET\_CONTROL**; see *Section 7.4.5*).
  - The Host Controller shall effectively propagate the global Abort signal to the PIO Queues (as well as to any other operating contexts, such as Scheduled Commands).
  - The Host Controller shall remain in **ABORTED** state until the Driver clears the global **ABORTED** state and the **ABORTED** state for the PIO Queues.
  - Once the Driver writes a value of 1'b0 to clear the **ABORT** bit in register **HC\_CONTROL**, the Driver shall also clear the **ABORTED** state of the PIO Queues by writing a value of 1'b0 to clear the **ABORT** bit in register **PIO\_CONTROL** (*Section 7.5.13*). After this, processing shall be resumed (if such Command Descriptors remain enqueued).

- 2678 • **Additionally, in DMA Mode:**
- 2679     • The Host Controller Abort operation acts as though the Ring Abort operation (see *Section 6.6.6*) were  
2680       triggered for all Ring Bundles that were currently valid (i.e., enabled and running) when the Host  
2681       Controller received the request:  
2682         • Each Ring Bundle that is affected by the Host Controller Abort operation shall report its state as  
2683           **ABORTED**, per *Section 6.6.6* and *Section 7.6.10.8*.  
2684         • Any other Ring Bundles that were not valid (i.e., not running, not enabled, or already in **ABORTED**  
2685           state) shall not be affected (i.e., shall not enter the **ABORTED** state).  
2686     • The Host Controller shall effectively propagate the global Abort signal to all such Ring Bundles (as  
2687       well as any other operating contexts, such as Scheduled Commands). Each affected Ring Bundle shall  
2688       enter the **ABORTED** state simultaneously, and each must be recovered individually.  
2689     • The Host Controller shall remain in **ABORTED** state until the Driver clears the global **ABORTED** state:  
2690         • Reads from bit **ABORT** in register **HC\_CONTROL** shall return a value of 1'b1 while the Host Controller  
2691           remains in the **ABORTED** state.  
2692         • Once the Driver writes a value of 1'b0 to clear the **ABORT** bit in register **HC\_CONTROL**, the Driver  
2693           shall also clear the **ABORTED** state for the Ring Bundles that entered the **ABORTED** state.  
2694         • The Driver shall follow the same procedure to perform Ring recovery steps, for each Ring Bundle in  
2695           **ABORTED** state (per *Section 6.6.6*).  
2696         • As each such Ring Bundle is recovered and brought out of **ABORTED** state, it shall become available  
2697           for Ring Bundle Servicing (per *Section 6.6.5*) if and only if the Host Controller was also brought out  
2698           of the global **ABORTED** state.  
2699     • **For any other Ring Bundles that were not Aborted:** If the Driver later enables such a Ring Bundle  
2700       (i.e., one that was not enabled when the Host Controller Abort operation was processed), or causes a  
2701       Ring Bundle that was not running to enter running state (i.e., one that was stopped when the Host  
2702       Controller Abort operation was processed), then:  
2703         • If bit **ABORT** in register **HC\_CONTROL** still has a value of 1'b1, then the Host Controller shall remain  
2704           in an **ABORTED** state, and no Ring Bundles shall be serviced (i.e., until this is first cleared at the Host  
2705           Controller level).  
2706         • If bit **ABORT** in register **HC\_CONTROL** was cleared (i.e., written with a value of 1'b0), then such a  
2707           Ring Bundle shall become available for Ring Bundle Servicing. No other action to clear any other  
2708           **ABORTED** state is necessary for this Ring Bundle, as it was not previously **ABORTED**.

Writing a value of 1'b1 to bit field **ABORT** in register **HC\_CONTROL** always serves as the method of initiating the global Abort request. The Driver shall follow up the specific action(s) appropriate for the current operating mode, as given in *Table 6*.

**Table 6 Actions and States for ABORT Bit in Register HC\_CONTROL**

Current Operating Mode	ABORT Bit Read Value	Status	Expected Next Action
<b>Scheduled Commands</b>	1'b1	Host Controller remains in global <b>ABORTED</b> state. Processing for Scheduled Commands is blocked.	Driver must write 1'b0 to this field to resume global operation, before re-enabling processing of Scheduled Commands ( <b>Section 6.16</b> ).
	1'b0	<b>ABORTED</b> state is cleared by Driver.	None
<b>PIO Mode</b>	1'b1	Host Controller remains in global <b>ABORTED</b> state. Processing for PIO Command Queue is blocked.	Driver must write 1'b0 to this field to resume PIO Command Queue processing
	1'b0	<b>ABORTED</b> state is cleared by Driver. Aborted PIO Queues may then be resumed and recovered.	Driver must then recover PIO Queues and clear <b>ABORTED</b> state, in register <b>PIO_CONTROL</b> ( <b>Section 7.5.13</b> ).
<b>DMA Mode</b>	1'b1	Host Controller remains in global <b>ABORTED</b> state. Processing for all Ring Bundles is blocked.	Driver must first write 1'b0 to this field to clear Host Controller <b>ABORTED</b> state.
	1'b0	Host Controller <b>ABORTED</b> state is cleared by Driver. All aborted Ring Bundles may then be resumed and recovered individually.	Driver must then recover affected Ring Bundles and clear <b>ABORTED</b> state for each, in register <b>RH_CONTROL</b> ( <b>Section 7.6.10.9</b> ).

### 6.8.5 Support for I3C ‘Short’ Read-Type Transfers

A Host Controller shall support ‘short’ Read-Type transfers, as defined in [Section 6.2.7](#) of the I3C TCRI Specification [[MIPI06](#)]. Additionally, the Host Controller shall support any Transfer Command types that allow the software to select whether to consider a ‘short’ Read-Type transfer as a transfer error.

**Note:**

*A ‘short’ Read-Type transfer might not necessarily be an error, as some I3C content protocols might depend on variable-length Private Read transfers. From the perspective of this I3C HCI Specification, it is left to the Driver or higher-level software to decide whether to (a) Indicate that ‘short’ Read-Type transfers will be treated as a transfer error by the Host Controller, and stop any processing of subsequent Transfer Commands; or (b) Indicate that ‘short’ Read-Type transfers that do not otherwise generate any other errors should not stop processing of subsequent Transfer commands. However, in case (b) the ultimate decision of whether or not the ‘short’ Read-Transfer was actually an error (i.e., unexpected result) is left to the Driver: in some cases, the Driver might inspect the Response data to decide whether this transfer was an error. If so, then the Driver could also initiate a Host Controller Abort operation to stop processing at some later time.*

### 6.8.6 Support for Transfer Termination and HDR Mode Configuration

A Host Controller shall optionally support special handling of I3C data transfers where either the I3C Target(s) or a Monitoring Device can terminate the transfer. If supported, then the Host Controller shall support configuration flows using Command Descriptors of type Internal Control Command (see [Section 8.4.2](#)) with value 0x6 in field **MIPI\_CMD** (see [Table 135](#) and [Table 141](#)). In this section and its subsections, such a Command Descriptor is referred to as a Configuration Command Descriptor.

#### 6.8.6.1 Use with HDR Mode Configuration

If the I3C Bus has one or more Targets that support special handling for Early Transfer Termination in one or more HDR modes, then the Host Controller allows software to configure such I3C Targets (using the **ENDXFER** CCC, per the I3C Specification [[MIPI02](#)] at [Section 5.1.9.3.25](#)). After such I3C Targets are configured appropriately, then software may construct a command sequence using a Configuration Command Descriptor followed by one or more Transfer Commands in the indicated HDR Mode.

To use this special handling, software shall first enqueue a Configuration Command Descriptor with a value of 0x1 (for HDR-DDR Mode) or 0x2 (for HDR-Ternary Modes) in field **END\_XFER\_TYPE** (i.e., Bits[15:12], see [Table 141](#)), and the configuration byte previously sent to such Targets with the **ENDXFER** CCC in field **END\_XFER\_CONFIG** (i.e., Bits[23:16]). This Configuration Command Descriptor temporarily configures the I3C Bus Controller Logic to support the provided configuration parameters for that HDR Mode, which shall apply for the rest of the command sequence. These configuration parameters vary per the HDR Mode, and may include:

- Whether the Controller and Target(s) use, or expect to handle, a particular Early Termination Pattern during the transfer, giving the receiver an opportunity to either continue or terminate the transfer;
- Whether the Controller or Target(s) will send any additional structured protocol elements after an Early Termination Pattern was initiated by the receiver (for both Writes and Reads); and
- Whether the Controller and Target(s) will use flow control for certain transfer types (i.e., an equivalent of ACK/NACK in that HDR Mode protocol).

As this Configuration Command Descriptor is always used in a command sequence, any subsequent Transfer Commands in that HDR Mode will be sent with the appropriate parameters, until the next STOP condition. To use special handling, software should send one or more Transfer Commands in the sequence, where the last such Transfer Command contains the value 1'b1 in the **TOC** field, and any previous Transfer Commands contain the value 1'b0 in the **TOC** field (i.e., continuous framing, as defined in [Section 6.12](#)). After exiting the HDR Mode with the HDR Exit Pattern (which ends with the STOP condition), the I3C Bus Controller Logic shall return to its default configuration parameters.

2757

**Note:**

2758 For HDR-DDR Mode and HDR-Ternary Modes, the special handling and Early Transfer Termination  
2759 support is new for v1.1 of the I3C Specification, and certain I3C Targets may choose to support  
2760 these capabilities. Since other Targets might not support these capabilities (including legacy I3C  
2761 Targets that only comply with version 1.0 of the I3C Specification), the Configuration Command  
2762 Descriptor only applies the provided configuration parameters to the current command sequence,  
2763 and the I3C Bus Controller Logic uses its default parameters for subsequent transfers (i.e.,  
2764 subsequent Transfer Commands after the STOP). This allows software to discover which I3C  
2765 Targets will support this special handling, optionally configure them with the **ENDXFER** CCC to use  
2766 special handling, and then enqueue command sequences for these I3C Targets, starting with the  
2767 Configuration Command Descriptor. However, software must first configure any I3C Targets with  
2768 the **ENDXFER** CCC before using the Configuration Command Descriptor that uses special  
2769 handling. When enqueueing command sequences for other I3C Targets (i.e., those that do not  
2770 support special handling, or those that were not configured to use it), software shall not use the  
2771 Configuration Command Descriptor in the command sequence.

#### 6.8.6.2 Use with Monitoring Device Early Termination

2772 If the I3C Bus has one or more Targets or Monitoring Devices that can terminate a transfer using  
2773 Monitoring Device Early Transfer Termination, then the Host Controller allows software to construct a  
2774 command sequence that enables this Early Termination capability for the I3C Bus, using a Configuration  
2775 Command Descriptor followed by one or more Transfer Commands in the indicated I3C Mode.

2776 To use this special handling, software shall first enqueue a Configuration Command Descriptor with a value  
2777 of 0x3 in field **END\_XFER\_TYPE** (i.e., Bits[15:12], see *Table 141*), and the configuration parameter byte in  
2778 field **END\_XFER\_CONFIG** (i.e., Bits[23:16]). This Configuration Command Descriptor will temporarily  
2779 configure the I3C Bus Controller Logic to support the Early Termination capability for the indicated I3C  
2780 Modes, which shall apply for the rest of the command sequence. Additionally, the Host Controller will  
2781 begin the SDR Frame by automatically sending the **ENDXFER** CCC with the same configuration parameter  
2782 byte, such that it enables all Monitoring Devices to use the parameters for subsequent transfers.

2783

**Note:**

2784 The value in field **END\_XFER\_CONFIG** is used directly by the I3C Bus Controller Logic, when it  
2785 sends the **ENDXFER** CCC with Defining Byte 0xFC to configure the I3C Bus.

2786 As this Configuration Command Descriptor is always used in a command sequence, any subsequent  
2787 Transfer Commands in that I3C Mode will be sent with the appropriate parameters, until the next STOP  
2788 condition. Once the I3C Bus Controller Logic is ready to use Early Transfer Termination, software should  
2789 send one or more Transfer Commands in the sequence, where the last such Transfer Command contains the  
2790 value 1'b1 in the **TOC** field, and any previous Transfer Commands contain the value 1'b0 in the **TOC** field  
2791 (i.e., continuous framing, as defined in *Section 6.12*). After ending transfers with either the STOP condition  
2792 (for SDR Mode) or the HDR Exit Pattern (for HDR Modes), the I3C Bus Controller Logic shall return to its  
2793 default configuration parameters.

## 6.9 IBI Handling

This Section specifies IBI Handling in PIO Mode and DMA Mode.

For both PIO Mode and DMA Mode, all IBI Data Bytes shall be stored as DWORDs in the current byte ordering mode (as indicated by field **DATA\_BYTE\_ORDER\_MODE** in register **HC\_CONTROL**, see [Section 7.4.2](#)).

### 6.9.1 IBI Handling in PIO Mode

In PIO Mode, the IBI-related information shall be enqueued to the IBI Queue and consumed by reading the IBI Port (see [Section 7.5.4](#)). IBI reception shall be indicated via the IBI Status Port Threshold Status bit (field **IBI\_STATUS\_THLD\_STAT**) in the Interrupt Status register (register **PIO\_INTR\_STATUS**, see [Section 7.5.9](#)).

The Driver shall first read out an IBI Status Descriptor structure from the IBI Port (register **IBI\_PORT**, see [Section 7.5.4](#)). Next, the Driver shall read consecutive data DWORDs from the IBI Port, using the **DATA\_LENGTH** field provided via the IBI Status Descriptor (see [Section 8.6](#)).

**Note:**

*Both the IBI Status Descriptor and the IBI Data DWORD(s) are read from the same IBI Port. IBI Data DWORD(s) shall be read from the IBI Port, according to the current byte ordering mode per [Section 6.8.3](#).*

For some IBIs, multiple IBI Status Descriptors might be needed to describe a single IBI event. If field **LAST\_STATUS** in the previous IBI Status Descriptor is set to 1'b0, then the Driver shall continue reading from the IBI Port to read the next IBI Status Descriptor.

In cases with multiple back-to-back IBI Requests, the Driver should provide appropriate values in fields **IBI\_STATUS\_THLD** and **IBI\_DATA\_SEGMENT\_SIZE** of register **QUEUE\_THLD\_CTRL** (see [Section 7.5.5](#)), in order to ensure that IBI reception is not blocked due to the IBI Queue becoming completely full before it can be read by the Driver via the IBI Port. The Driver should choose values that balance overall system latency and performance concerns, with knowledge of the size of the IBI Queue. The recommended values might vary, depending on the frequency of IBI Requests as well as the expected data payload length for IBIs from different I3C Target Devices.

**Note:**

*The Host Controller shall buffer incoming IBI payload data before populating the IBI Status Descriptor into the IBI Queue, since the IBI Status Descriptor contains the length of the data that follows it (i.e., as read from the IBI Queue via the IBI Port).*

*If the IBI Status Threshold value is set to a value that is too high for the Driver to be made aware of incoming IBI Status Descriptors in a timely manner, then the Driver might not receive notifications and would not know to drain the IBI Queue in time for subsequent IBI Requests. In this case, if the IBI Queue were to become completely full, then the Host Controller would not be able to accept an incoming IBI Request from an I3C Target Device, so in such a case the Host Controller shall NACK all IBI Requests (including Hot-Join Requests and Controller Role Requests) until the IBI Queue is drained by the Driver.*

### 6.9.2 IBI Handling in DMA Mode

2830 **Note:**

2831     *This section applies only for Host Controller implementations that implement DMA Mode, and thus*  
2832     *connect to a System Bus that supports memory-mapped IO and DMA.*

2833 In DMA Mode, after an IBI is fully processed on the Bus, the Host Controller shall trigger an  
2834 **IBI\_READY\_STAT** interrupt to the Driver. For an IBI from a Target that is present in the DAT table, the IBI  
2835 shall be enqueued into the Ring Bundle indicated by field **RING\_ID** in that Target's DAT table entry. For an  
2836 IBI from an unknown Target, the IBI shall be enqueued into Ring Bundle 0.

2837 Per **Section 6.6.3.1**, the IBI Status Ring's Enqueue Pointer indicates that one or more IBI Status Descriptor  
2838 structures have been populated on IBI Status Ring. The Driver shall process one or more of these IBI Status  
2839 Descriptor structures, and upon completion shall update the IBI Status Ring's Dequeue Pointer to notify the  
2840 Host Controller that the structure has been freed.

2841 If an IBI Status Descriptor structure also indicates that it has an allocation of one or more Data Chunks  
2842 containing IBI payload data, then the Driver shall also process the IBI payload data, and then shall notify  
2843 the Host Controller that the Data Chunks have been freed (see **Section 6.6.3.2**).

2844 **Note:**

2845     *IBI payload data shall be written into the Chunks comprising the IBI Data Ring, according to the*  
2846     *current byte ordering mode that was selected when the IBI was received and processed by the*  
2847     *Host Controller's I3C Bus Controller Logic per Section 6.8.3.*

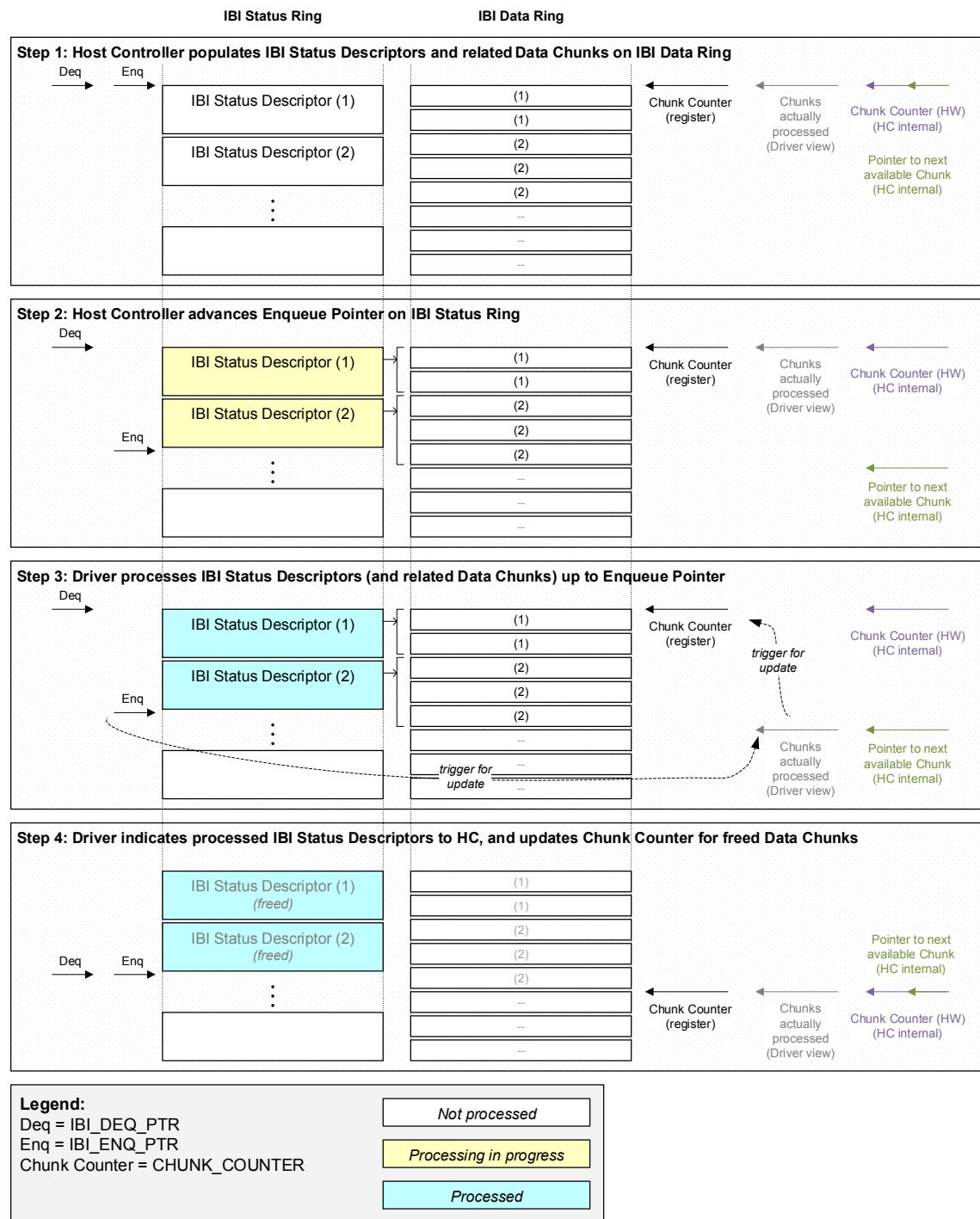
2848 The Driver shall also increment field **CHUNK\_COUNTER** in register **CHUNK\_CONTROL** (see **Section 7.6.10.3**)  
2849 to inform the hardware how many Data Chunks were freed (i.e., the Driver shall add the number of Data  
2850 Chunks for the processed IBI Status Descriptor(s) to the current value of field **CHUNK\_COUNTER**). This  
2851 approach relieves the Host Controller of the need to directly inspect the IBI Status Descriptors that the  
2852 Driver frees.

2853 To process one or more IBI Status Descriptor structures on the IBI Status Ring, along with any associated  
2854 Data Chunks on the IBI Data Ring, the Driver shall:

- 2855 1. Ensure that the Enqueue Pointer is ahead of the Dequeue Pointer
- 2856 2. Read the Chunk Counter field and save the cached value
- 2857 3. Read the first IBI Status Descriptor (i.e., the one at the position pointed to by the Dequeue Pointer)  
2858     from the IBI Status Ring
- 2859 4. Determine whether this is the last IBI Status Descriptor for an IBI event (i.e., check field  
2860     **LAST\_STATUS**)
- 2861 5. Determine how many Data Chunks are associated with this IBI Status Descriptor (i.e., check field  
2862     **CHUNKS**) and process these Data Chunks appropriately (i.e., copy contents of Data Chunks from the  
2863     IBI Data Ring into other Host system memory)
- 2864 6. Advance the Dequeue Pointer by one position
- 2865 7. Add the number of Data Chunks to the cached value of the Chunk Counter field, and write this new  
2866     value back to the Chunk Counter field
- 2867 8. Optionally repeat steps 2–7, as long as the Dequeue Pointer has not yet reached the Enqueue Pointer.

2868 The Driver may also optimize this procedure, by freeing several adjacent IBI Status Descriptors with one  
2869 update (i.e., advancing the Dequeue Pointer by multiple positions) and then freeing all Data Chunks for  
2870 such IBI Status Descriptors with a single write to field **CHUNK\_COUNTER** in register **CHUNK\_CONTROL**, as  
2871 long as it adds the total number of all Data Chunks for all such IBI Status Descriptors.

2872  
2873  
2874  
2875 **Figure 24** illustrates IBI Status Ring pointers and Chunk Counter field updates during IBI processing. This figure also shows the Host Controller's internal pointer that tracks the number of free Data Chunks, shown here as “Chunk Counter (HW)”. Note that “Chunk Counter (HW)” also necessarily tracks the Host Controller’s position in the IBI Data Ring (see **Section 6.6.3.1** and **Section 6.6.3.2**)



2876 **Figure 24 IBI Processing in DMA Mode**

### 6.9.3 Timestamping

The **TS** field of a DAT table entry indicates whether the corresponding Target supports Timestamping. If a Target supports Timestamping, then the Host Controller hardware shall include two additional QWORDs (thus 4 DWORDs in total) in the IBI Data buffer, containing Controller Timestamp information (**C\_REF** and **C\_C2**) for the IBI event on the I3C Bus. These Controller Timestamp counters are sent before the IBI Mandatory Byte, and the IBI Payload Length is increased by 4 DWORDs in order to accommodate them. The exact means of implementing **C\_REF** and **C\_C2** and its Synchronization to any System timer is implementation-specific and beyond the scope of this Specification.

**Note:**

*The Host Controller does not interpret or otherwise process any counters directly provided by any Target (**T\_C1** and **T\_C2**). The software shall handle correlation of **C\_REF** and **C\_C2** with **T\_C1** and **T\_C2**, based on the definitions given in the I3C Specification [MIPI02] and any additional private contract between the Controller and the Target.*

In PIO Mode, the **C\_REF** and **C\_C2** counters appear after the initial IBI Status Descriptor structure from the IBI Queue Port.

In DMA Mode, where the IBI Status Descriptor structure is put on the IBI Status Ring, the first chunk in the IBI Data Ring starts with the **C\_REF** and **C\_C2** counter values.

The IBI Status Descriptor structure indicates the presence of Controller timestamping counters in the IBI Data Queue or IBI Data Ring by setting the **TS** field to 1'b1. In cases where this IBI payload is spread across multiple chunks (for DMA Mode) and more than one IBI Status Descriptor structure is inserted into the IBI Status Ring, all such IBI Status Descriptor structures shall have the **TS** field set to 1'b1, but the Controller counters shall be coalesced just once at the beginning of the first portion of the IBI Data buffer (i.e., in the first chunk only for DMA Mode).

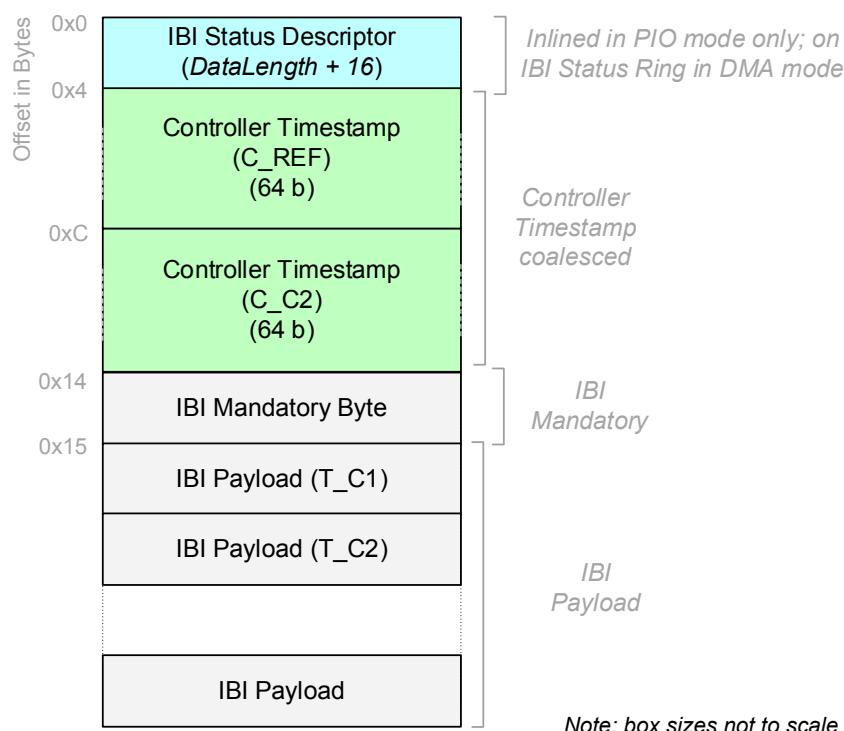


Figure 25 Controller Timestamp Counters for IBI Event

#### 6.9.4 IBI Data Abort Operation

If field **IBI\_DATA\_ABORT\_EN** in register **HC\_CAPABILITIES** (*Section 7.4.4*) has a value of 1'b1, then the Host Controller also supports the IBI Data Abort operation, which allows the Driver to abort reception of an ongoing IBI data payload.

The Driver may enable or disable the IBI Data Abort monitoring logic by setting or clearing bit **IBI\_DATA\_ABORT\_MON** in register **IBI\_DATA\_ABORT\_CTRL** (*Section 7.4.18*). When monitoring is enabled, the Host Controller shall watch for subsequent writes to this register with Bit[8] set to 1'b1. If an IBI is received (i.e., if it is accepted) by the Bus Controller Logic, then the Host Controller shall allow the Driver to abort the IBI data payload once the Host Controller has received the first IBI Status Descriptor. The Driver must provide matching field values when writing to this register, as follows:

- Field **MATCH\_IBI\_ID** must match the value provided in field **IBI\_ID** from the IBI Status Descriptor (i.e., the Target's Dynamic Address with **RnW** = 1'b1); and
- Field **MATCH\_STATUS\_TYPE** must match the value provided in field **STATUS\_TYPE** from the IBI Status Descriptor, with restrictions:
  - Only values 3'b000 (**REGULAR\_IBI**) and 3'b100 (**AUTOCMD\_READ**) are supported.

If the Host Controller sees a write to this register with the matching fields above, and if the same IBI data payload or Auto-Command Read is still in progress, then the I3C Bus Controller shall terminate the read at the next opportunity (i.e., at a byte boundary) and shall then send at least one final IBI Status Descriptor (with any other unsent data bytes). The last IBI Status Descriptor shall have field **LAST\_STATUS** set to 1'b1.

**Note:**

*Since HCI register writes are not synchronous, if the IBI Data payload or Auto-Command read transfer had previously ended before the Host Controller could process the requested IBI Data Abort operation, then the Host Controller shall take no additional action. Writes to the indicated fields of register **IBI\_DATA\_ABORT\_CTRL** shall not be 'sticky', and shall not affect any subsequent incoming IBIs.*

*If the Driver writes non-matching values to the indicated fields of register **IBI\_DATA\_ABORT\_CTRL**, then the Host Controller shall take no action.*

*The I3C Bus Controller Logic shall end the IBI data payload at the soonest opportunity (i.e., shall abort the transfer on a byte boundary), and shall report any additional bytes using IBI Status Descriptor(s) and additional DWORDs. If the IBI data payload is aborted, then at least one additional IBI Status Descriptor shall be generated, the last of which shall have field **LAST\_STATUS** set to 1'b1.*

**Note:**

*The Driver should monitor the incoming IBI Status Descriptor and data DWORDs as it determines whether to request the IBI Data Abort operation. This decision might depend on either the length of the IBI data payload, or whether the Driver matches a specific data pattern in the IBI data payload. Once the Driver writes to register **IBI\_DATA\_ABORT\_CTRL** and requests the IBI Data Abort operation, the Host Controller shall end the transfer in the proper manner for the current I3C Mode (for example, to end an IBI data payload in SDR Mode, the I3C Bus Controller Logic would use the T-Bit). Similar data transfer ending procedures shall be used for HDR Modes, for Auto-Command read transfers in any HDR Modes.*

*The IBI Data Abort operation is most useful for longer IBI data payloads or Auto-Command read transfers. Additionally, the Driver must configure the Host Controller to send quicker notification of the first IBI Status Descriptor, and then the Host Controller should send the successive data DWORDs at more frequent opportunities, which might not be the most efficient strategy if monitoring were not enabled. For example, in PIO Mode effective use of the IBI Data Abort operation depends on the IBI Queue threshold settings (i.e., register **QUEUE\_THLD\_CTRL**, see *Section 7.5.5*).*

### 6.9.5 Target IBI Credit Counting

If field **IBI\_CREDIT\_COUNT\_EN** in register **HC\_CAPABILITIES** (*Section 7.4.4*) has a value of 1'b1, then the Host Controller also supports an Target IBI credit counting mechanism, which allows the Driver to manage the credits for each Target. Target IBI credit counting is intended for use on I3C Buses where I3C Targets send frequent IBI Requests with data payloads, and when the Driver needs to manage incoming IBI Requests and ensure that it has sufficient memory to handle the IBI payload data.

When this mechanism is enabled, the Host Controller shall conditionally ACK or NACK a Target's IBI Request, based on the available credits for that Target in the Target IBI Credit Counters Extended Capability structure (see *Section 7.7.5*). If the Host Controller supports this mechanism, then it shall also include this Extended Capability structure. Each Target's credit counter is stored in a register within the Extended Capability structure, and the Host Controller shall consume credits (i.e., shall decrement the counter) as data bytes are received. All Target credit counters shall be cleared when the Host Controller is reset, or when the Target IBI credit counting mechanism is disabled (i.e., when the Driver clears field **R/S** in register **TARGET\_CREDIT\_CONFIG**, see *Section 7.7.5.2*).

**Note:**

*The Extended Capability structure has an array of registers, with one register for each Target. The size of the array is the same as the number of DAT entries (see *Section 8.1*), and each register for a Target corresponds to its DAT entry (i.e., has the same positional index).*

*When this mechanism is enabled, the Host Controller shall ignore field **IBI\_REJECT** in the Target's DAT entry. Conversely, when this mechanism is disabled, the Host Controller shall use field **IBI\_REJECT** in the DAT entry to determine whether to accept or reject the IBI Request.*

As each IBI Request arrives, the Host Controller checks the credit counter for that Target, to see whether the IBI Request is allowed. If there is not sufficient credit, then the Host Controller shall NACK the IBI Request, and shall then immediately send the **DISEC** Direct CCC with the '**DISINT**' bit set, to tell the Target to disable subsequent IBI Requests.

**Note:**

*Per the I3C Specification [**MIPI05**] at *Section 5.1.6.2*, the Controller may NACK an IBI Request and also send the **DISEC** CCC to disable sending subsequent IBI Requests. I3C Targets are allowed to send IBI Requests by default, unless configured to not send them via the **DISEC** CCC (per [**MIPI05**] *Section 5.1.9.3.1*). However, as noted in I3C FAQ [**MIPI08**] entry *Q18.20*, disabling IBI Requests only affects an I3C Target's ability to send IBI Requests, it does not prevent an I3C Target from enqueueing new IBI Requests that would be sent a later time, as and when the Controller sends the **ENEC** CCC to re-enable sending IBI Requests.*

However, if there is sufficient credit when the IBI Request is received, then the Host Controller shall ACK the IBI Request, and shall prepare to receive the IBI data payload and optional Auto-Command data (e.g., Pending Read). As the data arrives, the Host Controller shall automatically decrement the credit counter based on the length of the data received, per the value of field **CREDIT\_BYTE\_COUNT** in register **TARGET\_CREDIT\_CONFIG**. One credit shall be consumed for the IBI Request and its Mandatory Data Byte, followed by one credit for every **K** bytes of additional IBI payload data or subsequent Pending Read payload data, where the value **K** is determined by 2 to the power of (1 plus the value in field **CREDIT\_BYTE\_COUNT**).

**Note:**

*If the IBI data payload or Pending Read data payload is not exactly aligned to the byte count for a credit, then the Host Controller shall decrement the credit counter by 1 for the last portion of the data payload, i.e., the partial byte count. If an IBI Request has a data payload of either zero bytes (i.e., no MDB) or one byte (i.e., MDB only), then the Host Controller shall decrement the credit counter by 1.*

If the Target's credit counter reaches zero during (or at the very end of) an IBI Request or a subsequent Pending Read, then the Host Controller shall automatically send the **DISEC** Direct CCC with the '**DISINT**'

2995 bit set after the end of the data payload, to tell the Target to disable subsequent IBI Requests. The Host  
2996 Controller shall conditionally terminate the read if the credit counter reaches zero, per the value of field  
2997 **TERM\_READ\_ZERO\_CREDIT** in register **TARGET\_CREDIT\_N** (where **N** is the index of both the register array  
2998 and the Target's DAT entry).

2999 **Note:**

3000 *If field **TERM\_READ\_ZERO\_CREDIT** is set to 1'b0, then the Host Controller shall continue reading  
3001 data bytes from the Target even if the credit counter reaches zero during the read. The Driver  
3002 should prepare to read up to the maximum expected length of the IBI data payload (or subsequent  
3003 Pending Read data payload); in most cases, this requires the Driver to have sufficient memory to  
3004 handle the data payload that would exceed the allotted credit count.*

3005 *If field **TERM\_READ\_ZERO\_CREDIT** is set to 1'b1, then the Host Controller shall terminate the read  
3006 of the data payload when the credit counter reaches zero (i.e., if the Target did not use the T-Bit to  
3007 indicate the end of the payload). The Host Controller shall indicate this condition by setting field  
3008 **ERROR** to 1'b1 in the last IBI Status Descriptor.*

3009 After the credit reaches zero, the Driver should subsequently increment the credit counter if it wishes the  
3010 Host Controller to accept subsequent IBI Requests from that Target. The Driver may update the available  
3011 credit by writing to register **TARGET\_CREDIT\_N** in one of two ways:

- 3012 • Incrementing the credit counter, when field **CREDIT\_INCR** is set to 1'b1, and field **CREDIT\_COUNT** holds  
3013 the number of credits to increment.
  - 3014 • If the value written to field **CREDIT\_COUNT** would cause the Target's credit counter to exceed the  
3015 maximum value, then the Host Controller shall constrain this to the maximum value. The maximum  
3016 value is a function of the value of field **CREDIT\_MAX\_SIZE** in register **TARGET\_CREDIT\_TABLE** (see  
3017 *Section 7.7.5.1*).
  - 3018 • If the Target's credit counter was previously zero when the Host Controller applied the update, then the  
3019 Host Controller shall automatically send the **ENECC** Direct CCC with the '**ENINT**' bit set, to tell the  
3020 Target to enable IBI Requests.
- 3021 • Decrementing the credit counter, when field **CREDIT\_DECR** is set to 1'b1, and field **CREDIT\_COUNT** holds  
3022 the number of credits to decrement.
  - 3023 • If the Target's credit counter is already zero, then the Host Controller shall discard the update.
  - 3024 • If the value written to field **CREDIT\_COUNT** would cause the Target's credit counter to go below zero,  
3025 then the Host Controller shall constrain this to zero. The Host Controller shall also automatically send  
3026 the **DISECC** Direct CCC with the '**DISINT**' bit set, to tell the Target to disable IBI Requests.

3027 **Note:**

3028 *The Driver shall not set both bits **CREDIT\_INCR** and **CREDIT\_DECR** to 1'b1 when writing to register  
3029 **TARGET\_CREDIT\_N**. If the Host Controller sees both of these bits set during a register write, then it  
3030 shall discard the update.*

3031 *If fields **CREDIT\_INCR** and **CREDIT\_DECR** are both set to 1'b0, then the Host Controller shall not  
3032 update the Target's IBI credit counter, and shall generate an IBI credit acknowledgement report that  
3033 contains the current value of the credit counter.*

3034 As the Host Controller processes the credit update request, it will first wait for the IBI credit update FSM to  
3035 reach an idle state (i.e., no IBI Request is being received; see *Figure 26* below), ensure there is space in the  
3036 IBI Queue/Ring to enqueue the acknowledgement report (see below), and then apply the increment or  
3037 decrement operation to the current value in the Target's credit counter. After applying the update, the Host  
3038 Controller shall generate an IBI credit acknowledgement report that is a special form of an IBI Status  
3039 Descriptor (see *Section 8.6.4*) that contains the status of the update.

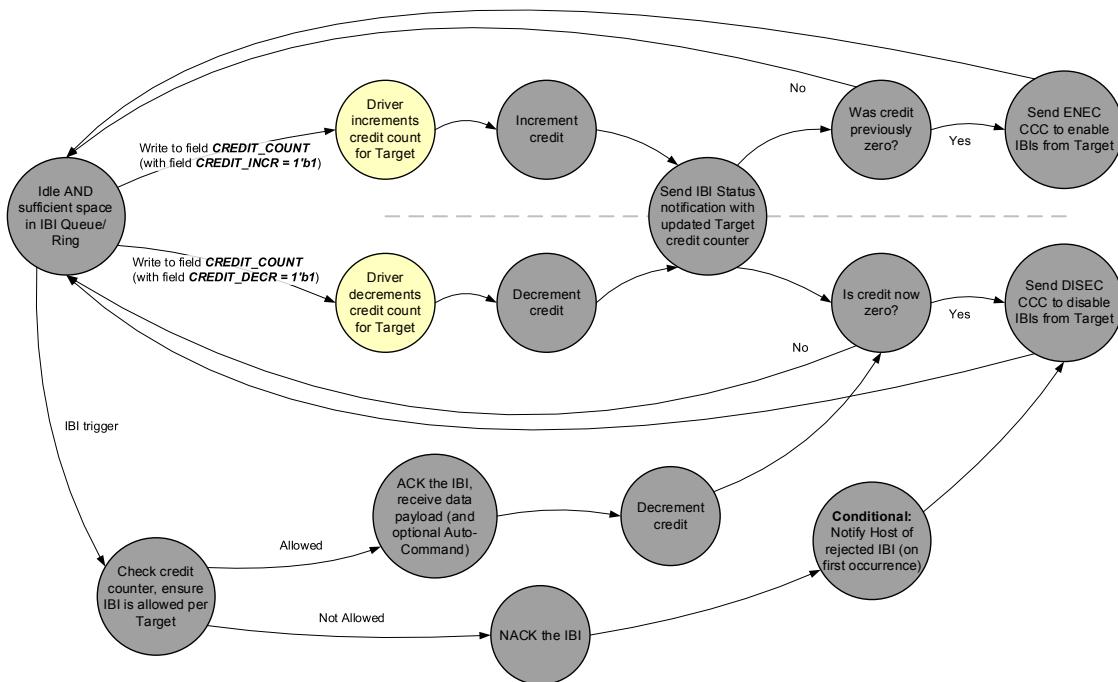
3040 **Note:**

3041 *If the IBI Queue (for PIO Mode) or IBI Ring (for DMA Mode) is full when the Host Controller sees  
3042 the register write to update the credit, then the Host Controller shall delay applying the update until  
3043 there is sufficient space to enqueue the IBI Status Descriptor and the single DWORD. When  
3044 sufficient space is available, the report shall be enqueued into the IBI Queue/Ring with highest  
3045 priority.*

3046  
3047  
3048  
3049  
3050  
3051

The credit mechanism only supports one pending credit update request at a time, and the Host Controller shall delay applying the request if the IBI credit update FSM is not currently in the idle state or if the IBI Queue/Ring is full (as noted above). If a new credit update request is received while a pending request is already delayed and cannot yet be applied, then the Host Controller shall drop the new request, and eventually generate an IBI Status Descriptor with a value of 1'b1 in field **IBI\_STS**, to report that the new request was dropped.

3052 The Host Controller shall use the FSM shown in **Figure 26**, to manage updates to a Target's credit counter.



**Figure 26 Target Credit IBI Update FSM**

3053  
3054  
3055  
3056  
3057  
3058  
3059

If the Host Controller sends the **DISEC** Direct CCC in response to a Target that tries to raise an IBI Request that is not allowed, then the Host Controller shall also conditionally generate an IBI Status Descriptor that indicates the IBI was rejected, per the value of field **NOTIFY\_IBI\_REJECTED** in register **IBI\_NOTIFY\_CTRL** (see **Section 7.4.17**). If this field is set to 1'b1 (**ENABLED**), then the Host Controller shall only do this once per Target while its credit counter is zero, and shall not send additional IBI Status Descriptors for this reason unless the Target's credit counter is subsequently incremented and then decremented to zero.

3060 **Note:**

3061 When the Host Controller automatically sends the **DISEC** CCC with the '**DISINT**' bit set in response  
3062 to the credit counter reaching zero or an IBI Request that is not allowed, it should do as soon as  
3063 possible. In cases where the IBI Request is NACKed, the Host Controller shall drive a Repeated  
3064 START immediately after the NACK and shall then send the **DISEC** CCC. In cases where the credit  
3065 counter reaches zero and the IBI data payload must be terminated (per the value of field  
3066 **TERM\_READ\_ZERO\_CREDIT**), the Host Controller shall terminate the read transfer, and then drive a  
3067 Repeated START followed by the **DISEC** CCC. This ensures that the Target can apply the new  
3068 configuration as soon as possible.

## 6.10 Managed CCC Transfer Framing Model

The Host Controller supports the Managed CCC Transfer Framing model defined in *Section 6.3* of the I3C TCRI Specification [*MIPI06*] and its sub-sections. Additional specifications in this section address details particular to specific HCI operating modes.

### 6.10.1 Direct CCCs and NACK Retry

The DAT allows the Driver to configure the NACK retry count for typical Transfer Commands to particular I3C Targets. However, this is handled differently for Direct CCCs.

**When processing a Transfer Command that is a Direct CCC:**

**If the indicated Device NACKs its Dynamic Address along with the RnW bit, then:**

- The Host Controller shall attempt to retry the Direct CCC at least once following the first NACK, per the following conditions:
  - If the value of the appropriate field **DEV\_NACK\_RETRY\_CNT** (i.e., from the indicated Device's DAT entry) indicates no retries for NACKed transfers, then the Host Controller shall always attempt one retry for a NACKed Direct CCC, per the I3C Specification's mandatory single-retry model (see the I3C Specification at *Section 5.1.9.2.3 [MIPI02]*).
  - However, if the value of the appropriate field **DEV\_NACK\_RETRY\_CNT** indicates any additional retries, then the Host Controller shall retry according to that field's setting.
- If the indicated Device NACKs its Dynamic Address repeatedly, beyond the retry count determined by the above conditions, then this shall be reported as an error per *Section 6.10.3*.

### 6.10.2 Providing Data for Broadcast/Direct Write CCCs

When enqueueing Transfer Commands for CCCs that write data to a Target, the software shall observe the following requirements.

**For PIO Mode:** The Data bytes for Broadcast, Direct Write, or Direct SET CCC segments shall be populated on the Data Queue as a sequence of DWORD writes, in an appropriate order for each Command Descriptor not using Immediate Data Bytes. Software should generally attempt to write all the Data to the Data Queue before writing any Command Descriptors to the Command Queue (see *Section 6.8.1*). All such Command Descriptors should be written to the Command Queue in sequential order.

**For DMA Mode:** Each Command Descriptor shall be contained in a Transfer Descriptor structure (see *Section 8.3*). For Direct Write or Direct SET CCC segments, the relevant Data pointer for each Transfer Descriptor structure shall point to the Data buffer for the corresponding Target Device. If using multiple Transfer Commands for CCCs sent in continuous framing, then the first Transfer Descriptor shall point to the Data for the first Target, the second Transfer Descriptor structure shall point to the Data for the second Target, etc. If using Broadcast CCCs, then each Transfer Descriptor structure shall point to the data for that Broadcast CCC; these may optionally be mixed with other Direct CCC segments, or be mixed with other Broadcast CCCs, or be sent individually. All of the Transfer Descriptors that form such a CCC transfer shall be posted to same Ring Bundle, and shall be contiguous (i.e., shall not be interleaved with other Command Descriptors). Software should generally prepare and enqueue these Transfer Descriptors in one operation, before updating the Enqueue Pointer (see *Section 6.8.2*).

For additional details, refer to *Section 6.3.1.1* of the I3C TCRI Specification.

### 6.10.3 Error Handling for CCC Flows

As the Host Controller processes each Command Descriptor, it shall generate and provide Response Descriptors appropriately, as indicated in field **WROC** for each Command Descriptor, and each Response Descriptor shall provide status information for the related transfer phase.

- **For PIO Mode:** Response Descriptor structures shall be generated for all Command Descriptors with field **WROC** having a value of 1'b1, for all Direct Read or Direct GET CCCs (i.e., as with any Read-type transfer), or when the transfer phase encountered an error for any Command Descriptor (see *Section 6.5*).
- **For DMA Mode:** Multiple Response Descriptor structures are generated in the Response Ring, one for each Command Descriptor, with status in each descriptor (see *Section 6.6.2*).

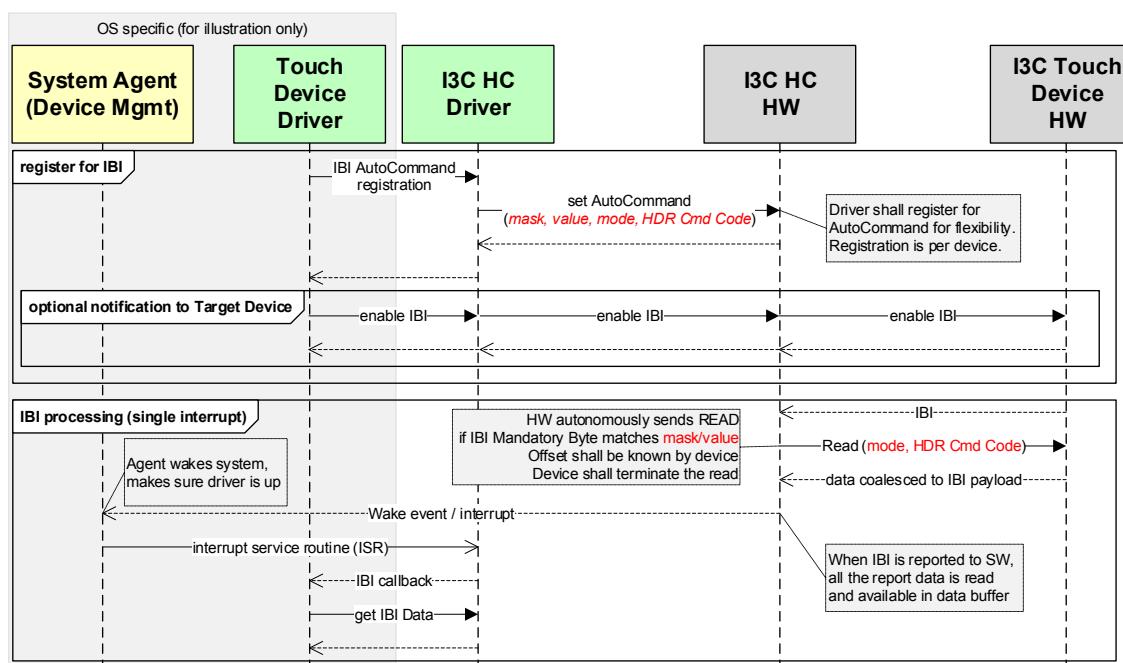
## 6.11 Auto-Command

The Host Controller supports an option to automatically generate a Read transfer after receiving a specified IBI event on a per-Device (per-Target) basis. If enabled, this option requires setting what value of the IBI Mandatory Byte will trigger this Auto-Command.

**Note:**

*The Auto-Command capability corresponds to the Pending Read Notification contract that is defined in the I3C Specification (see [MIP105] at Section 5.1.6.2.2).*

**Figure 27** illustrates the high-level flow for Auto-Command triggering. Note that the optional OS-specific part (shown as grayed-out), if present, it is entirely OS-implementation-specific. Once the Auto-Command is enabled, the Host Controller shall inspect IBI Mandatory Byte coming from the indicated Device (Target). The IBI Mandatory Data Byte is masked with the value set by the Driver.



**Figure 27 Auto-Command Generation Flow**

### 6.11.1 Flow of Operation

When the Host Controller receives an IBI that is accepted from the indicated Device (i.e., the Target Address in the IBI matches the **DYNAMIC\_ADDRESS** field in the DAT entry) and whose IBI Mandatory Byte, after masking with the **AUTOCMD\_CMD** field of the DAT entry for this Device, matches the indicated **AUTOCMD\_VALUE**, the Host Controller shall immediately initiate a Read-type transfer to this Device (i.e., before allowing another IBI or another Transfer/Command to proceed). The Read-type transfer shall have the parameters indicated by fields in the DAT entry: field **AUTOCMD\_MODE** shall specify the I3C Mode, and field **AUTOCMD\_HDR\_CODE** shall specify the Command Code for HDR Modes (e.g., HDR-DDR, HDR-TSP, and HDR-TSL). The Device is expected to know the internal offset (if any) and data length for the Read data, and shall handle proper termination of the transfer.

After the termination of the transfer, the Host Controller shall store the received data into the IBI Payload Buffer, and the Driver shall process the IBI only once all the Data becomes ready. The Driver shall process the request in exactly the same manner as though the data had been received via a regular IBI event (i.e., as the IBI Data Payload).

### 6.11.2 Configuration with DAT

In order to enable the Auto-Command feature, the software shall:

1. For all Devices for which Auto-Command behavior is desired, follow the DAT entry and:
  - A. Set field **IBI\_PAYLOAD** to 1'b1 to allow reception of IBI Mandatory Byte
  - B. Set field **AUTOCMD\_MASK** to the desired value
  - C. Set field **AUTOCMD\_VALUE** to the desired value
  - D. Set field **AUTOCMD\_MODE** to the desired value
  - E. Set field **AUTOCMD\_HDR\_CMD** to the desired value

Every time an IBI occurs for an Auto-Command enabled Device, the IBI's received Mandatory Data Byte is masked with the value of **AUTOCMD\_MASK** and the result is logically compared to the value of **AUTOCMD\_VALUE**. If the compare result is True, then the Host Controller automatically performs a Read-type transfer to the indicated Device after the completion of the IBI, using the provided parameters and in the I3C Mode described with field **AUTOCMD\_MODE**. The read transfer is unbounded in length, and thus ends when terminated by the Target Device, or when terminated by the Controller in case of an end-of-buffer condition (i.e., for an IBI Data Ring or an IBI FIFO).

**Example:** Assume that **AUTOCMD\_MASK** contains 0xF and that **AUTOCMD\_VALUE** contains 0xC. IBIs with the following Mandatory Data Byte values will trigger the Auto-Command: 0x0C, 0x1C, ..., 0xFC.

**Note:**

*The default values of **AUTOCMD\_MASK** and **AUTOCMD\_VALUE** are both 0x00, and when **IBI\_PAYLOAD** is set to 1 in the DAT, an Auto-Command shall be triggered for all incoming IBIs from the indicated Dynamic Address. If Auto-Command is not intended, then the software must update field **AUTOCMD\_VALUE** to a non-zero value in order to disable the Auto-Command function.*

The Read command in HDR Mode shall use 0x80 as the default Command Code, in case the value in **AUTOCMD\_HDR\_CODE** is lower than 0x80, or the value defined with **AUTOCMD\_HDR\_CODE**. See *Table 65* of the I3C Specification [[MIP102](#)] for details.

### 6.11.3 Data Reporting to IBI Payload Buffer

Field **AUTOCMD\_DATA\_RPT** in register **HC\_CONTROL** (see *Section 7.4.2*) determines how the data received from this Read transfer will be reported:

- **Coalesced Reporting:** If field **AUTOCMD\_DATA\_RPT** is set to 1'b0, then data received from this Auto-Command's Read transfer is coalesced together with the IBI Data Payload, and must be described using the same IBI Status Descriptor structure(s), per *Section 8.6.2*.

In this case, the Read transfer data byte(s) from the Auto-Command read will immediately follow the last data byte from the IBI Data Payload. The Driver must know the length of the IBI Data Payload, which may vary per the I3C content protocol. For all IBI Status Descriptors, field **STATUS\_TYPE** will contain the value 3'b000 (**REGULAR\_IBI**).

- **Separated Reporting:** If field **AUTOCMD\_DATA\_RPT** is set to 1'b1, then data received from this Auto-Command's Read transfer is described in separate IBI Status Descriptor structure(s) that immediately follow the IBI Status Descriptor(s) and IBI Data for the IBI Data Payload, per *Section 8.6.6*.

In this case, the initial IBI Status Descriptor(s) for the IBI Data Payload will have field **STATUS\_TYPE** with a value of 3'b000 (**REGULAR\_IBI**). For the subsequent IBI Status Descriptor(s) for the Auto-Command Read transfer data, field **STATUS\_TYPE** will contain the value 3'b100 (**AUTOCMD\_READ**).

**Note:**

*Field **AUTOCMD\_DATA\_RPT** is new for version 1.2 of this I3C HCI Specification, and the IBI Descriptor's field **STATUS\_TYPE** has been expanded to allow more status types, including the*

3181      *separated Auto-Command Read transfer data (see **Section 8.6** for more details). By contrast,*  
3182      *version 1.1 and earlier of this I3C HCI Specification only supported coalesced reporting.*

## 6.12 Request Handling

In general, software shall not initiate a transfer request to an I3C Target Device until and unless the Target has been assigned a Dynamic Address per **Section 6.4**.

- The Address Assignment Command type of the Command Descriptor (see **Section 8.4.1**) shall be used to perform Dynamic Address Assignment using either **ENTDAA** or **SETDASA**.
- A standard Transfer Command (such as an Immediate Transfer Command) shall be used to perform Dynamic Address Assignment using **SETAASA**.

Once a Target Device has received its Dynamic Address, the software may initiate transfer requests for any supported Transfer Command types. This includes Transfer Commands that are CCCs using managed CCC transfer framing (per **Section 6.3** of the I3C TCRI Specification [**MIPI06**]) in SDR Mode. Each Command Descriptor shall indicate the Target Device to which the Transfer Command is directed.

**Note:**

*Per **Section 6.3** and **Section 6.4**, the Driver must first populate a DAT entry containing the Dynamic Address for a Device (or an assigned Group Address, if supported) before sending any transfer requests using the index for that DAT entry.*

The Speed and Mode of the transfer is set via the **MODE** field in the Command Descriptor structure for a Transfer Command type (for more details, see **Section 6.2.5** of the I3C TCRI Specification [**MIPI06**]).

Per **Section 6.2.4** of the I3C TCRI Specification [**MIPI06**], Software may enqueue individual transfers, where each Command Descriptor's field **TOC** has a value of 1'b1; or a number of transfers in a given order, comprising a transaction sequence:

- The first and subsequent Command Descriptors (except for last in the sequence) indicate the start and continuation of the sequence, with field **TOC** having a value of 1'b0; and
- The last Command Descriptor indicates the end of the sequence, with field **TOC** having a value of 1'b1.

Within such a sequence, field **TOC** roughly corresponds with the framing elements in the I3C Modes, as defined in I3C TCRI Specification **Section 6.2.4**.

**Note:**

*Unless otherwise noted in this I3C HCI Specification, software should generally not mix Transfer Commands (as defined in the I3C TCRI Specification) with other such Command Descriptors that are not Transfer Commands, especially in the same sequence. Specifically, other such Command Descriptor(s) should not be inserted in the middle of a valid sequence of Transfer Commands with field TOC set to 1'b0, because the results would be undefined. Instead, software should ensure that the last Transfer Command has field TOC set to 1'b1, or else ensure that the Bus has otherwise reached an idle state (i.e., due to transfer abort or some other error) before enqueueing a Command Descriptor that is not a Transfer Command. However, certain Atomic transaction sequences that are defined in this I3C HCI Specification will require an Internal Control Command at the start and end of such a sequence, in order to inform the I3C Bus Controller Logic of special handling of such Transfer Commands.*

Whenever possible, the software should attempt to enqueue all Command Descriptors comprising a sequence of transfers as a single continuous operation or uninterrupted stream of actions, in order to prevent a command sequence stall condition or timeout (as defined in the I3C TCRI Specification at **Section 6.4.2** [**MIPI06**]). If the Command Queue (for PIO Mode) or the active Command Ring (for DMA Mode) is unable to contain or accept all such Command Descriptors for this sequence, or if the sequence is longer than the maximum size of the Command Queue/Ring, then software must first enqueue some Command Descriptors to start processing the sequence, and then actively monitor the status of the Command Queue/Ring to ensure that it is able to enqueue additional Command Descriptors when possible, while preventing a command sequence stall condition or timeout.

For more detail, refer to **Section 6.4.2** of the I3C TCRI Specification [**MIPI06**].

The Host Controller shall monitor the active Command Queue/Ring and attempt to detect situations of possible command sequence stall conditions or timeouts, reporting them as warnings or errors per **Section 6.13.2**. The Host Controller shall also monitor the specific execution status of the operating mode, for other mode-specific conditions that might cause an interruption of transfer processing.

Whenever possible, the Host Controller and its I3C Bus Controller Logic should attempt to mitigate or prevent such underflow situations that might become imminent as the Command Queue drains to approach an empty state, using the following methods as examples:

- Stalling the I3C Bus, using any methods that might be defined and enabled for the I3C Mode;
- Using any available I3C-Mode-specific methods to defer a transaction, or any upcoming actions that might be expected in the near future; or
- Reducing the clock speed at which transfers are driven on the Bus.

In other cases, the Host Controller and its I3C Bus Controller Logic would not be able to prevent an underflow situation or other interruption of transfer processing, and would be required to cancel the sequence. In this case, the I3C Bus Controller Logic would be required to end the framing using either a STOP condition or HDR Exit Pattern, as appropriate for the I3C Mode. As a result, the I3C Bus Controller Logic would act as though the last executed Command Descriptor had been provided the value 1'b1 in field **TOC** (instead of its actual value of 1'b0).

If the Host Controller is instructed to cancel a transaction sequence for any reason (including a command sequence timeout) then it shall assert an interrupt via field **HC\_SEQ\_CANCEL\_STAT** in register **INTR\_STATUS** (see **Section 7.4.7**) to inform the Host that the sequence was terminated before receiving a Command Descriptor having field **TOC**=1. The specific warning and error conditions relating to a command sequence stall condition or timeout shall also be reported as interrupts, as defined in **Section 6.13.2**.

**Note:**

*Since some I3C content protocols might not tolerate a forced STOP condition between two transfers that would otherwise need to be executed in a continuous sequence (i.e., with a Repeated START or HDR Restart Pattern), the Driver shall determine whether enqueueing the next Command Descriptor after a cancelled sequence due to a timeout is a correct course of action, or whether the command sequence must be restarted from an earlier transfer. The Driver shall determine whether to resume transfers after the STOP condition, or to restart from a prior Transfer Command, by enqueueing new Transfer Commands appropriately for the I3C content protocol after detecting the forced STOP condition. Alternatively, the Driver may configure automatic halting if a command sequence timeout occurs, by writing to field HALT\_ON\_CMD\_SEQ\_TIMEOUT in register HC\_CONTROL (see **Section 6.13.2** and **Section 7.4.2**).*

Refer to **Section 6.7** and **Section 8.4** for usage of Transfer Commands.

A Host Controller may support alternate methods for handling a transition between two Transfer Commands with different Speeds (i.e., different transfer rates) or different I3C Modes (see **Section 6.2.5** of the I3C TCRI Specification [**MIPI06**]), or for optimizing the handling of Transfer Commands in HDR Modes with field **TOC** set to 1'b1. If such optional alternate methods are supported, then they shall be disabled by default, and shall be explicitly enabled by software via register control. Such a register should be part of an Extended Capability structure, that might be enabled or disabled by software, having a default configuration setting to use the standard behavior.

If the Host Controller processes either a Host Controller Abort operation (see **Section 6.8.4**) or a Ring Abort Operation (see **Section 6.6.6**) at the request of the Driver, then it shall attempt to either complete or abort any in-progress transfers that the I3C Bus Controller Logic is currently driving, and if possible it shall abort such transfers at the earliest opportunity, even if the associated Command Descriptor(s) have field **TOC** set to 1'b0 (i.e., are not the intended end of a sequence).

### 6.12.1 PIO Mode

In PIO Mode, the Driver prepares the Command Descriptor structure with the Request, and optionally the associated data buffer (see [Section 6.8.1](#)). The Data is sent to the Tx Data Port register 32 bits (one DWORD) at a time (see the PIO section of the Register Map, [Section 7.4.21](#)). The Data is transmitted on the I3C Bus for each DWORD entry in the Tx Data Queue, using the current byte ordering mode (per [Section 6.8.3](#)). Each Data byte is transmitted on the I3C Bus according to the I3C Mode signaling. The Driver shall provide data payload and write the Command Descriptor to the Command Queue Port, starting with the first DWORD entry.

The order in which the data and the Command Descriptor are provided may be arbitrary (i.e., Command Descriptor first, or Command Descriptor last). Once the last DWORD of the Command Descriptor is put into the Command Queue, the Host Controller shall start processing the Transfer Command. In PIO Mode, the data can be sent either as an Immediate transfer payload (i.e., in a Command Descriptor structure), or else through the Transfer Data Port.

Per [Section 6.5](#), once the transaction has taken place on the Bus, the Host Controller shall conditionally populate the Response Queue with a Response Descriptor (see [Section 8.5](#)) describing the status of this transaction. For Read-Type transfer requests, the data for the transaction is available through the Transfer Data Port. Each Data byte is populated from the Bus based on the I3C Mode signaling. The software shall populate the data into the Rx Data Queue as one or more DWORD entries, using the current byte ordering mode (per [Section 6.8.3](#)). If the received data length is not DWORD-aligned, then the next Bus transaction shall be populated starting on a DWORD boundary.

If any error is detected, then the PIO processing shall be halted and the Response Descriptor shall indicate the particular error. After reporting the error, status shall be cleared ([PIO\\_INTR\\_STATUS](#)) and written to the Command Queue Port ([COMMAND\\_QUEUE\\_PORT](#)).

If the software is still enqueueing Command Descriptors that are part of a sequence of transfers (as defined in [Section 6.12](#)) but has not yet enqueued the last such Command Descriptor for the end of this sequence (i.e., where field **TOC** would have a value of 1'b1) for any reason, this might cause a command sequence stall condition or timeout, if the Host Controller is able to process all previously enqueued Command Descriptors before the software is able to enqueue a subsequent Command Descriptor into the Command Queue Port. Such a stall condition shall initially be indicated as a warning, per the I3C Mode. This stall condition would be mitigated if the Host Controller were to receive a subsequent Command Descriptor in time, i.e., if software were able to enqueue a subsequent Command Descriptor via the Command Queue Port (see [Section 6.8.1](#)) before the stall becomes a timeout. However, if the Host Controller were forced to cancel the transaction sequence due to a timeout caused by an empty Command Queue, then it must report this timeout error as an interrupt, as defined in [Section 6.13.2](#); and it must also indicate the cancellation of the transaction sequence, via field **HC\_SEQ\_CANCEL\_STAT** of register [INTR\\_STATUS](#) (see [Section 7.4.7](#)).

**Note:**

*If the Host Controller is required to cancel a transaction sequence due to other PIO error conditions such as the Tx Data Queue draining to empty during a Write-Type transfer, or the Rx Data Queue becoming completely full during a Read-Type transfer, and this causes the transfer to be aborted by the I3C Bus Controller Logic, then this shall also be reported as a cancelled transaction sequence, in addition to an aborted transfer (i.e., field **TRANSFER\_ABORT\_STAT** in register [PIO\\_INTR\\_STATUS](#); see [Section 7.5.9](#)).*

For each Command Descriptor in a sequence, if field **TOC** has a value of 1'b1, then the Bus transaction shall end with a STOP condition, in order to allow subsequent IBI reception. If field **TOC** has a value of 1'b0, then the Bus transaction shall end with a Repeated START condition, and the next Command shall be processed.

### 6.12.2 DMA Mode

In DMA Mode, the Driver prepares pointers to one or more contiguous buffers of physical memory, depending on whether Scatter-Gather capability is supported (per [Section 6.2.1](#)). The Command Descriptor structure is contained in a Transfer Descriptor structure at a given index on the Command Ring. The Transfer Descriptor also indicates whether to store the data in a single contiguous block of physical memory, or in a list of memory blocks (i.e., an array of Memory Descriptor structures) using Scatter-Gather capability (if supported). The flow for preparing a Transfer Descriptor is defined in [Section 6.8.2](#), and the data structure formats are defined in [Section 8.3](#).

Once software has written the Transfer Descriptor's pointer into the Command Ring and has updated the Enqueue Pointer, the Transfer Descriptor is ready to be processed. Software may enqueue one or more Transfer Descriptors with a single update of the Enqueue pointer.

The I3C Bus Controller Logic shall use the current byte ordering mode (per [Section 6.8.3](#)) for all Transfer Commands. For Write-Type transfers, the Memory Access Interface shall read the indicated DWORDs automatically from Host system memory and apply the current byte ordering mode to each DWORD, to ensure that the Data bytes enter the Tx Data Buffer in the correct order. Similarly, for Read-Type transfers, the Memory Access Interface shall consume bytes from the Rx Data Buffer and collect these the Data bytes into DWORDs accordingly, to ensure that every DWORD written to Host system memory uses the current byte ordering mode.

The byte ordering shall apply to Transfer Commands that use a single physical memory buffer, as well as Transfer Commands that use Scatter-Gather (i.e., to each single buffer described by a Buffer List Pointer list entry; see [Section 8.3.1](#)).

The Memory Access Interface logic shall drive all Host system memory access for Transfer Commands using the DMA engines (i.e., Memory Access Engines, as shown in [Figure 3](#)), using the Base Address pointers that are provided by the Host for each Ring, as well as the buffer pointer for each Transfer Descriptor. The Ring Controller shall monitor the fill levels of the Tx Data Buffer and Rx Data Buffer, and also monitor the operation of the Memory Access Interface for all outstanding memory requests. The Ring Controller shall detect situations where the Host system bus is unable to respond to memory requests with sufficient performance to maintain correct operation of the I3C Bus Controller Logic during a transfer, and where the I3C Bus Controller Logic might be required to terminate a transfer.

For each transaction, the index into the Command Ring and Response Ring shall be the same. That is, at the same Ring index for both the Command Ring and the Response Ring, the Host Controller shall conditionally generate a Response Descriptor for each Transfer Descriptor that is processed and executed, and the Response Descriptor at that index shall indicate the status of processing that transaction, per [Section 6.6.2](#). Response Descriptors shall be generated as transactions are processed, according to the Ring Controller's arbitration logic, which might service individual Ring Bundles in any desired order or priority. Additionally, the Ring Bundle must be enabled and running, and not in a halted or aborted state.

If the software is still enqueueing Transfer Descriptors that are part of a sequence of transfers (as defined in [Section 6.12](#)) but has not yet enqueued the last such Transfer Descriptor for the end of this sequence (i.e., where the Command Descriptor's field **TOC** would have a value of 1'b1) for any reason, then this might cause a command sequence stall condition or timeout if the Host Controller is able to process all previously enqueued Transfer Descriptors in this Ring Bundle before the software is able to enqueue a subsequent Transfer Descriptor into the Command Ring of this same Ring Bundle. Such a condition shall initially be indicated as a warning, per the I3C Mode. This situation would be mitigated if the Host Controller receives a "doorbell" notification that one or more subsequent Transfer Descriptors were written into Command Ring (see [Section 6.8.2](#)) before the stall becomes a timeout. However, if the Host Controller must cancel the transaction sequence due to an "empty" Command Ring (i.e., when the Enqueue Pointer and Dequeue Pointer are equal), then it would have to report this timeout error as an interrupt, as defined in [Section 6.13.2](#); and it would also indicate the cancellation of the transaction sequence, via field **HC\_SEQ\_CANCEL\_STAT** of register **INTR\_STATUS** (see [Section 7.4.7](#)).

## 6.13 Error Handling

The general approach to error handling in I3C HCI is as follows:

- Any detectable error on a transaction will result in an error code in the Response Descriptor structure.

Reception of the Response triggers an interrupt to Host Controller Driver.

In addition, the Host Controller shall halt processing to allow the Driver to handle the error.

- The specific error code will be returned in field **ERR\_STATUS** of the Response Director, as defined in **Section 8.5**. **Section 6.4** of the I3C TCRI Specification [**MIPI06**] contains additional details on the defined error codes in field **ERR\_STATUS**.

- In **PIO Mode**, the Command/Response Queue shall halt its operation (indicated via the **RESUME** field in register **HC\_CONTROL**, **Section 7.4.2**). The Driver shall take any necessary measures to recover from this situation, and then resume normal operation of the queues.

- In **DMA Mode**, the Ring Bundle that processed the failed request shall halt its operation (indicated via the RS field in register **RH\_CONTROL**, **Section 7.6.10.9**). The Driver shall take any necessary measures to recover from this situation, and then resume normal operation of that Ring Bundle.

In some error situations, the Host Controller might halt operations of all enabled Ring Bundles, indicated via the **RESUME** field in register **HC\_CONTROL**. The Driver shall take any necessary measures to recover from this situation, and then resume normal operations of applicable Ring Bundles.

- For Write requests, the Driver should issue a Transfer Command with the **GETSTATUS** Direct CCC to determine the Device's overall error status.

- IBIs (Target-initiated interrupts) are natively supported via the IBI Ring and triggering interrupt (i.e., the same as for the Response Descriptor). Any errors in IBI reception are flagged in the IBI Status Descriptor structure.

The Host Controller shall not autonomously issue the **GETSTATUS** Direct CCC, nor any other command to determine Device state/health. The Driver shall issue **GETSTATUS** Direct CCC when required, and shall subsequently take appropriate measures to recover the Bus or the Host Controller.

If the Host Controller exposes Debug-specific registers via the Debug Specific Extended Capability structure (*see Section 7.7.7*), the Host Controller may optionally provide a counter for CE2 errors via field **CE2\_ERROR\_COUNT** in register **MX\_ERROR\_COUNTERS** (**Section 7.7.7.4**). Such error counters may optionally be used by software to indicate the type of the errors occurring on the Bus. Any such registers shall zero their values upon being read; as a result, any cumulative counter must be implemented in software.

Upon any non-recoverable internal error, the Host Controller shall stop, interrupt the software, and indicate an internal error via field **HC\_INTERNAL\_ERR\_STAT** in register **INTR\_STATUS** (**Section 7.4.7**).

### 6.13.1 Error Status Codes in Response Descriptor

General error codes for Response Descriptors are defined in *Section 6.4.1* of the I3C TCRI Specification [*MIPI06*]. This I3C HCI Specification defines the specific conditions under which reporting of some of those error codes is required. Some of these conditions depend upon the HCI operating mode or condition.

As a result, the following sub-sections should be read in conjunction with the I3C TCRI Specification's error code definitions.

#### 6.13.1.1 Address Header Error

For field **ERR\_STATUS** having a value of 0x4 (**ADDR\_HEADER**):

- **In SDR Mode:**

If no Target Devices acknowledge the Broadcast Address before the start of a private Transfer Command that was driven automatically by the I3C Bus Controller Logic:

- Because field **IBA\_INCLUDE** in register **HC\_CONTROL** (*Section 7.4.2*) was set to 1'b1,
- Or because software had previously used an Internal Control Command to enable automatic transmission of the I3C Broadcast Header on the Bus (per *Section 8.4.2* and *Table 137*),

then this is an Address Header error.

For additional details, see the I3C TCRI Specification at *Section 6.4.1.4 [MIPI06]*.

#### 6.13.1.2 Overflow/Underflow Error

For field **ERR\_STATUS** having a value of 0x6 (**OVL**):

- **In PIO Mode:**

If the Host Controller's Tx Data Buffer experiences an underflow condition for any Write-Type transfer, then this is an Underflow error (see *Section 6.8.1*).

If the Host Controller's Rx Data Buffer experiences an overflow condition for any Read-Type transfer, then this is an Overflow error.

**Note:**

*This scenario does not apply to DMA Mode, since the Host would allocate memory for the Tx/Rx Data Buffer; therefore, the Host does not use PIO registers to fill the Tx Data Queue or consume from the Rx Data Queue during a transfer.*

- **While processing a sequence of Command Descriptors for an Atomic transaction for a Target Reset Action:**

The Host Controller shall report this error code for the special Internal Control Command that indicates the end of the sequence, if the Atomic transaction was only partially successful.

“Only partially successful” means that the Command Queue/Ring had an underflow condition before the end of the sequence (i.e., that software could not enqueue all such Command Descriptors in a timely manner, to maintain continuous framing and avoid a timeout), and that some of the Transfer Commands in the sequence were successfully processed and driven on the I3C Bus.

In this scenario, the Host Controller would have determined that it could safely allow the partial sequence of Transfer Commands to be followed by the Target Reset Pattern, instead of allowing the stall to become a timeout, as defined in *Section 6.15.1.2*.

If the Command Queue/Ring underflow condition occurs and individual Transfer Commands in the sequence were rejected (i.e., were not driven on the I3C Bus due to underflow), then the Response Descriptors for each such rejected Transfer Command shall report error code 0x8 (**HC\_ABORTED**).

### 6.13.1.3 Host Controller Aborted

For field **ERR\_STATUS** having a value of 0x8 (**HC\_ABORTED**):

- **In PIO Mode:**

The Host Controller shall report this error code for a transaction that was aborted due to the Host Controller Abort operation per *Section 6.8.4*.

- **In DMA Mode:**

The Host Controller shall report this error code for a transaction that was aborted due to either the Host Controller Abort operation per *Section 6.8.4*, or a Ring Abort operation per *Section 6.6.6*.

### 6.13.1.4 Bus Aborted

For field **ERR\_STATUS** having a value of 0x9 (**BUS\_ABORTED**), the Host Controller shall report this error code if an I3C transaction was aborted due to Early Termination (i.e., by another I3C Device). This can occur when a Monitoring Device on the I3C Bus aborts a transfer, or when the intended Target uses an Early Termination Pattern to end a Write-Type transfer. For more details, see *Section 6.8.6*.

### 6.13.1.5 Aborted with CRC

The Host Controller does not support field **ERR\_STATUS** having a value of 0xB (**ABORTED\_WITH\_CRC**) because HDR-BT Mode transfers are not currently supported.

### 6.13.1.6 Transfer Type Specific

For field **ERR\_STATUS** having a value of 0xC–0xF (**Transfer Type Specific**):

These error codes are used for special variants of Transfer Commands, or in other operating modes.

- **While processing a Combo Transfer Command:**

For details that apply to Combo Transfer Commands, see the I3C TCRI Specification at *Section 6.4.1.2 [MIPI06]*.

- **While processing a sequence of Command Descriptors for an Atomic transaction for a Target Reset Action:**

The Host Controller shall report error code 0xC or error code 0xD for the special Internal Control Command that indicates the end of the sequence:

- If the Atomic Target Reset transaction failed due to early cancellation by software, or
- If the Command Queue/Ring had an underflow condition before the end of the sequence (i.e., if software could not enqueue all such Command Descriptors in a timely manner, to maintain continuous framing and avoid a timeout), as defined in *Section 6.15.1.2*.

In both cases, the Target Reset Action would not be performed as requested.

If the Command Queue/Ring underflow condition occurs, then any individual Transfer Commands that were rejected (i.e., that were not driven on the I3C Bus due to underflow) shall report error code 0x8 (**HC\_ABORTED**). For other transfer errors, the usual error codes shall apply to each individual Transfer Command within the sequence.

For other general guidance on Atomic transaction sequences, see the I3C TCRI Specification at *Section 6.4.1.2 [MIPI06]*.

- **While attempting to pass the Controller Role to a Secondary Controller Device:**

The Host Controller shall report error code 0xC or error code 0xD, depending on which error condition the I3C Bus Controller Logic encountered while attempting to pass the Controller Role with the **GETACCCR** CCC. For additional details, see *Section 6.17.2*. Note that a NACK error with the **GETACCCR** CCC is reported via error code 0x5 (**NACK**).

### 6.13.2 Errors Due to Command Sequence Stall or Timeout

In some situations, the Host Controller might fully drain the Command Queue (in PIO Mode) or active Command Ring (in DMA Mode) while processing a command sequence where one or more Command Descriptors were enqueued and the last Command Descriptor was a Transfer Command with field **TOC** set to 1'b0. As the I3C Bus Controller Logic started processing these enqueued Transfer Commands, the Host might not enqueue another suitable Command Descriptor with field **TOC** set to 1'b1 in a timely manner, and as a result the I3C Bus Controller Logic would be forced to terminate the sequence.

- In SDR Mode, the I3C Bus Controller Logic might stall the Bus at certain points in SDR framing, according to the maximum allowed stall times defined for SCL Low (see the I3C Specification [**MIPI02**] at *Section 5.1.2.5*).
- In HDR Modes, the I3C Bus Controller Logic might use a stall method that is specific to that HDR Mode, if supported and previously enabled for use.
  - If the stall method was not previously enabled or no stall method is supported, then the I3C Controller must terminate the transaction with the HDR Exit Pattern.

**Section 6.4.2** of the I3C TCRI Specification [**MIPI06**] defines handling requirements for Command Sequence Stall and Timeout.

In addition, this I3C HCI Specification defines the following Host Controller requirements:

- **When a Stall is Detected:** If the I3C Bus Controller Logic detects such a condition that it has temporarily mitigated by stalling the clock, then the Host Controller shall report this as a warning event, via field **HC\_WARN\_CMD\_SEQ\_STALL\_STAT** in register **INTR\_STATUS** (see *Section 7.4.7*).

If the condition is not resolved, and the stall turns into a timeout, then the Host Controller shall report this as a possible error, via field **HC\_ERR\_CMD\_SEQ\_TIMEOUT\_STAT** in register **INTR\_STATUS** (see *Section 7.4.7*). Software may configure the Host Controller to also assert an interrupt via field **HC\_SEQ\_CANCEL\_STAT** in register **INTR\_STATUS** (see *Section 7.4.7*) to inform the Host that the command sequence was terminated before receiving a Command Descriptor having field **TOC** = 1.

- **After a Timeout:** Software may subsequently resume sending valid Command Descriptors after the timeout, and the Host Controller shall attempt to restart a new transaction in the indicated I3C Mode, starting with the next Transfer Command in the sequence, unless any of the following conditions are true:
  - The Response Descriptor for the previous transfer (i.e., the last one processed before the stall turned into a timeout) indicated a transfer error in field **ERR\_STATUS**;
  - The Host Controller was processing Transfer Commands in an Atomic Target Reset transaction that had a timeout per **Section 6.15.1.2**, and would reject any subsequent Transfer Commands after a timeout, until software sends a special Internal Control Command to end this transaction mode;
  - Field **HALT\_CMD\_SEQ\_TIMEOUT** in register **HC\_CONTROL** (see *Section 7.4.2*) was set to 1'b1; or
  - Any other transfer error, operating mode-specific error or Host Controller interrupt was asserted.

3511 Field **HALT\_ON\_CMD\_SEQ\_TIMEOUT** in register **HC\_CONTROL** shall determine whether the Host Controller  
3512 must halt processing of further Command Descriptors in the active Command Queue/Ring, when the I3C  
3513 Bus Controller Logic encounters a command sequence timeout condition that it cannot avoid or mitigate.

- 3514 • If this field is set to 1'b1, then the Host Controller must halt processing and wait for software to resolve  
3515 the condition using an error recovery procedure. Software may set this field to 1'b1 if it knows that it is  
3516 sending sequences of Command Descriptors that must be run in sequence using continuous framing in  
3517 order to be successful, i.e., for a particular I3C content protocol that does not tolerate a disruption (i.e.,  
3518 forced end of the framing) if certain successive transfers are to be successful.

3519 For example, in cases where an unresolved stall becomes a timeout that causes a STOP condition or  
3520 HDR Exit Pattern, this would disrupt the operation of the addressed Target(s), or have other side  
3521 effects on the operation of addressed Target(s) that require subsequent commands to be continuously  
3522 executed with any preceding commands. As instructed by field **HALT\_ON\_CMD\_SEQ\_TIMEOUT**, the  
3523 Host Controller must not execute any subsequent commands after the forced STOP condition or  
3524 HDR Exit Pattern.

- 3525 • If this field is set to 1'b0, the Host Controller may generally resume processing without waiting for  
3526 software to resolve the condition. Software may set this field to 1'b0 if it is sending standard Transfer  
3527 Commands that may be executed separately or together, with the use of continuous framing as a  
3528 performance optimization.

3529 In this case, an unresolved stall that becomes a timeout shall still be reported as an error, but the Host  
3530 Controller shall resume operation upon receiving the next enqueued Command Descriptor, unless the  
3531 Host Controller was processing a special sequence of Transfer Commands that did not allow  
3532 automatic resume of operations, such as an Atomic Target Reset transaction.

3533 **Note:**

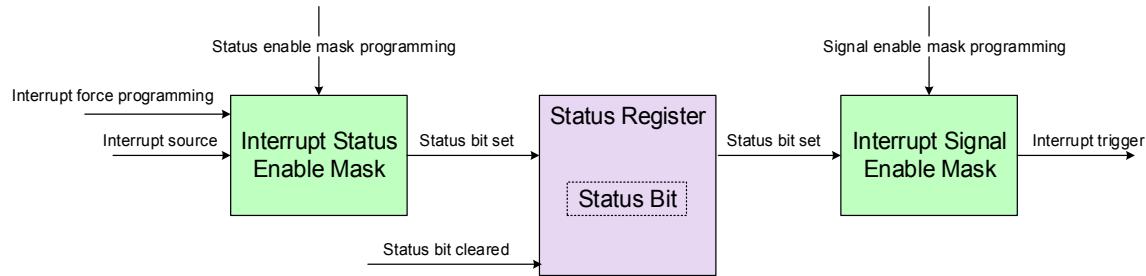
3534 *The Atomic Target Reset transaction using the **RSTACT** CCC and Target Reset Pattern (as defined  
3535 in **Section 6.15.1.2**) is a special exception with pre-defined behavior that always applies,  
3536 regardless of the current setting of field **HALT\_ON\_CMD\_SEQ\_TIMEOUT**.*

## 6.14 Interrupts

All interrupt-related Register Sets shall include registers for:

- Interrupt Status Register,
- Interrupt Status Enable Mask,
- Interrupt Signal Enable Mask, and
- Force Interrupt

**Figure 28** illustrates the logical dependencies among these interrupt-related registers.



**Figure 28 Logical Dependencies Among Interrupt-Related Registers**

Within each such Register Set, the corresponding bit fields for the four registers shall control the status and signaling of each interrupt condition. The bit fields in each of the four registers shall interact and provide fine-grained control over the reporting of interrupts to the Host system.

- The Interrupt Status Enable Mask bit field controls whether either of the following events might be logged in the Interrupt Status Register bit field:
  - The interrupt condition in logic, caused by the underlying event or assertion; or
  - Software writing a 1 to the Force Interrupt bit field (i.e., in the Interrupt Force register).
- If the bit field in the Interrupt Status Register has a value of 1, then it may be cleared either by taking some other action from software (i.e., by writing 1 to clear the same bit field), or by consuming the results of the specific Queue or FIFO that caused the underlying interrupt condition to trigger.
- The Interrupt Signal Enable Mask bit field controls whether the Interrupt Status Register bit field transitioning from 0 to 1 would cause the Host Controller to assert an interrupt to the Host.

In order to receive interrupt assertions for desired interrupt conditions, software must enable the bit fields in both the Interrupt Status Enable Mask and the Interrupt Signal Enable Mask. Alternatively, software might only enable bit fields in the Interrupt Status Enable Mask, and use polling of the Interrupt Status Register to test for an interrupt condition (i.e., without using an interrupt service routine).

The interrupt-related Register Sets shown in **Table 7** shall use this model.

**Table 7 Interrupt Register Sets**

Interrupt Register Set Purpose	Count	See Sections
General Host Controller errors or other actions	1 per HC	<b>7.4.7 through 7.4.10</b>
PIO Queue events or other PIO errors ( <i>if PIO Mode is supported</i> )	1 per HC	<b>7.5.9 through 7.5.12</b>
Ring Bundle events or other Ring errors ( <i>if DMA Mode is supported</i> ) (one Register Set for each supported Ring Header)	1 per Ring Header	<b>7.6.10.4 through 7.6.10.7</b>

3562 Certain interrupt-related Register Sets shall also report various warning and error events that signal  
3563 potential configuration issues, or warnings that might become errors. Some warning events in these  
3564 registers shall also be paired with a corresponding error event, which might also cause the Host Controller  
3565 to stop processing subsequent commands, as per the nature of the error event. In most cases, a warning  
3566 event might indicate an issue of a shorter duration, that would become an error if allowed to continue  
3567 without some form of mitigation or avoidance, usually by software. Software should enable these interrupt  
3568 conditions and read the warning events to understand cases of marginality or occasional system  
3569 performance issues.

## 6.15 Target Reset and Bus Recovery

The Host Controller shall support several methods of sending a Target Reset Pattern, or recovering the I3C Bus from various conditions.

**Note:**

*The reset and recovery methods described in this section are required in order to comply with version 1.1.1+ of the I3C Specification [MIPI05]. These methods rely on new sub-commands for the Internal Control Command (see Section 8.4.2). Implementers are required to support these sub-commands as well as the flows defined in sub-sections of this section.*

### 6.15.1 Target Reset Pattern

I3C version 1.1.1 supports several flows for the Target Reset mechanism (see the I3C Specification [MIPI02] at Section 5.1.11). The Host Controller shall support these flows using Command Descriptors of type Internal Control Command (see Section 8.4.2) with value 0x4 in the **MIPI\_CMD** field (see Table 135 and Table 139). In this section and its subsections, such a Command Descriptor is referred to as a Target Reset Command Descriptor.

#### 6.15.1.1 Reset I3C Peripheral or Reset Whole Chip

Software may choose to send the target Reset Pattern at any appropriate time by enqueueing a Target Reset Command Descriptor with a value of 0x0 in field **RESET\_OP\_TYPE** (i.e., Bits[13:12], see Table 139).

Software may issue a single Target Reset Pattern to initiate the Reset I3C Peripheral action, by sending one such Command Descriptor. Software may also issue two Target Reset Patterns to initiate the Reset Whole Chip action, by sending two such Command Descriptors in succession (i.e., without any intervening CCCs).

The Host Controller shall schedule the Target Reset Pattern to be sent once the Command Descriptor is processed, and once the I3C Bus is not active (i.e., a Bus Free Condition). The I3C Bus Controller Logic shall drive a START condition and set up the Bus properly, then drive the Target Reset Pattern, followed by a Repeated START and then a STOP.

#### 6.15.1.2 Configure Target for Target Reset Action

Software may choose to configure one or more Target Devices to take any valid Target Reset actions, using the Broadcast or Direct SET forms of the **RSTACT** CCC (see the I3C Specification [MIPI02] at Section 5.1.11.2 and Section 5.1.9.3.26). Software shall send a sequence of Command Descriptors to the Host Controller as an Atomic Target Reset transaction:

- Software shall first prepare the Host Controller to start the Atomic Target Reset transaction, by enqueueing a Target Reset Command Descriptor with field **RESET\_OP\_TYPE** (i.e., Bits[13:12]) set to a value of 0x2 (see Table 139).
  - This operation instructs the Host Controller to enter a Critical Section, such that it must perform special handling of subsequent Command Descriptors, until this is ended by a subsequent operation.
- Software shall then send one or more subsequent Command Descriptors in SDR Mode, with continuous framing. Each Command Descriptor shall be a Transfer Command:
  - A Broadcast CCC, addressed to the I3C Bus, to configure all Target Devices; or
  - A Direct SET CCC, addressed to a specific Target Device, for the Target Reset action that will be configured with the **RSTACT** CCC.

- 3606 • The last of these Command Descriptors shall contain the value 1'b1 in the **TOC** field, to indicate the end  
3607 of the Atomic transaction.
  - 3608 • To prepare a single Target Reset action, software shall send one such Command Descriptor with value  
3609 1'b1 in the **TOC** field. The Host Controller shall conditionally drive the Target Reset Pattern at the end  
3610 of the frame.
  - 3611 • To prepare multiple Target Reset actions, software shall send multiple consecutive Command  
3612 Descriptors in a sequence. Each Command Descriptor in the sequence except the last one shall contain  
3613 the value 1'b0 in the **TOC** field, and the last one shall contain 1'b1 in the **TOC** field. The Host Controller  
3614 shall use Direct CCC Framing (see *Section 6.10*) as indicated by the field values of the Command  
3615 Descriptors, and shall conditionally drive the Target Reset Pattern at the end of the frame (i.e., at the  
3616 end of the last such Command Descriptor, when **TOC**=1).
- 3617 • After sending all such Command Descriptors for the **RSTACT** CCC, software shall enqueue a Target  
3618 Reset Command Descriptor with field **RESET\_OP\_TYPE** (i.e., Bits[13:12]) set to a value of 0x3 (see *Table*  
3619 **139**).
  - 3620 • This operation instructs the Host Controller to leave the Critical Section and resume normal handling  
3621 of subsequent Command Descriptors.

3622 While the Host Controller is processing the Atomic transaction in the Critical Section, all other operations  
3623 shall be suspended. If the Host Controller is operating in DMA Mode, then any other active Ring Bundles  
3624 shall be temporarily suspended once the Host Controller starts processing the initial Target Reset Command  
3625 Descriptor, and then resumed once the entire transaction is finished (i.e., after the final Target Reset  
3626 Command Descriptor is received).

3627 The Host Controller shall observe the following requirements while operating in the Critical Section:

- 3628 • All Transfer Commands shall be in SDR Mode, at any supported data rate per *Section 7.1.1.1* of the I3C  
3629 TCRI Specification [**MIPI06**].
  - 3630 • The Host Controller shall reject any Transfer Commands in any HDR Modes.
  - 3631 • The Host Controller shall reject any other Internal Control Commands other than the Target Reset  
3632 Command Descriptor to leave the Critical Section.
  - 3633 • If the Host Controller receives the Target Reset Command Descriptor to leave the Critical Section, and it  
3634 has not yet received a valid Transfer Command for a RSTACT CCC with **TOC**=0, then:
    - 3635 • If this is intentional from software (i.e., is not due to a stall that turns into a timeout) then the Host  
3636 Controller shall cancel the Critical Section, and the I3C Bus Controller Logic shall not drive the Target  
3637 Reset Pattern.
    - 3638 • The Host Controller shall resume processing any subsequent Command Descriptors as normal, and  
3639 then generate a Response Descriptor with error code 0xD (**Transfer Type Specific: SEQ\_CANCELLED**) in  
3640 field **ERR\_STATUS** to signal that the Target Reset was cancelled by software request.
  - 3641 • The Transfer Commands allowed in the Critical Section shall be a sequence comprised of Broadcast  
3642 CCCs and/or Direct SET CCCs, for the **RSTACT** CCC.
    - 3643 • For certain sequences that begin with a single Broadcast **RSTACT** CCC (with Defining Byte of 0x0)  
3644 followed by one or more Direct SET **RSTACT** CCCs (with non-zero Defining Byte values), a ‘partial  
3645 success’ sequence is a truncated portion of the sequence, starting with the first (i.e., Broadcast with  
3646 Defining Byte 0x00) followed by one or more of the subsequent Direct SET **RSTACT** CCCs.
  - 3647 • The Host Controller shall inspect each Transfer Command received, after entering the Critical Section:
    - 3648 • If a Broadcast CCC is received as the first Transfer Command, then:
      - 3649 • If the Defining Byte is zero, then the Host Controller shall set a flag that indicates it may allow a  
3650 “partial success”.
      - 3651 • If the Defining Byte is not zero, then the Host Controller shall not allow a “partial success”.
    - 3652 • If no Broadcast CCC is received, then the Host Controller shall not allow a “partial success”.
    - 3653 • If a subsequent Broadcast CCC is received with a non-zero Defining Byte, and the “partial success”  
3654 flag was set earlier, then the “partial success” flag is cleared and shall not be set again during this  
3655 Critical Section.

- 3656     • If any Transfer Command fails during the sequence, then:
- 3657        • For failures due to a Target Device not acknowledging its Dynamic Address (i.e., **ERR\_STATUS** having  
3658            a value of 0x5 for **NACK**), the Host Controller shall stop processing subsequent Command Descriptors,  
3659            per normal processing, after attempting an appropriate number of retries. For Direct CCCs, the retry  
3660            model is specified in *Section 6.10*.
- 3661        • For failure due to any other reason, the Host Controller shall stop processing subsequent Command  
3662            Descriptors, per normal Transfer Command processing. The I3C Bus Controller Logic shall drive a  
3663            STOP condition, and the Target Reset Action shall be cancelled.
- 3664        • The Host Controller shall not drive the Target Reset Pattern, and shall leave the Critical Section due to  
3665            the error.
- 3666        • Software should subsequently run an error recovery procedure.
- 3667     • If the Host Controller processes all Command Descriptors that were enqueued into the Command  
3668            Queue/Ring, and the Host does not enqueue additional Command Descriptors in time to prevent a stall  
3669            from turning into a timeout, then:
- 3670        • If the “partial success” flag was set earlier, then the last processed Command Descriptor had field  
3671            **TOC**=0, and the Host Controller was able to drive the initial Broadcast CCC followed by at least one  
3672            subsequent Direct CCC (with **TOC**=0) in continuous framing, then the Host Controller shall assume it  
3673            could achieve success with a partially successful transaction.
- 3674        • For such a condition: before allowing the stall to turn into a timeout, the I3C Bus Controller Logic  
3675            shall drive the target Reset Pattern after the last successful Direct SET CCC Segment (i.e., the one  
3676            with the last processed Command Descriptor with **TOC**=0) followed by a Repeated START and then  
3677            a STOP.
- 3678        • The Host Controller shall report the warning via field **HC\_WARN\_CMD\_SEQ\_STALL\_STAT** of register  
3679            **INTR\_STATUS** (*Section 7.4.7*) to inform the Host that the I3C Bus Controller Logic was forced to  
3680            stall the clock (per *Section 6.13.2*) but prevented the command sequence stall condition from  
3681            becoming a timeout by driving the Target Reset Pattern before the end of the sequence.
- 3682        • The Host Controller shall set a flag indicating that all subsequent Command Descriptors in the  
3683            sequence would be rejected, until it receives the final Target Reset Command Descriptor with field  
3684            **RESET\_OP\_TYPE** set to the value 0x3.
- 3685        • Any Command Descriptor that is rejected (i.e., one that arrived after the I3C Bus Controller Logic  
3686            was forced to drive the Target Reset Pattern instead of allowing a stall to turn into a timeout) shall  
3687            cause a Response Descriptor to be generated, with error code 0x8 (**HC\_ABORTED**) in field  
3688            **ERR\_STATUS**.
- 3689        • When the Host Controller receives the final Target Reset Command Descriptor to leave the Critical  
3690            Section, it shall resume processing any subsequent Command Descriptors as normal, and then  
3691            generate a Response Descriptor with error code 0xC (**Transfer Type Specific: CMD\_UNDERFLOW**) in  
3692            field **ERR\_STATUS** to signal underflow of the Command Queue/Ring with partial success.

- 3693     • If the “partial success” flag was **not set** earlier, then the Host Controller shall **not** drive the Target  
3694       Reset Pattern, and shall allow the stall to turn into a timeout.  
3695
  - 3696           • The I3C Bus Controller Logic shall drive a STOP condition.
  - 3697           • The Host Controller shall assert an interrupt via the **HC\_SEQ\_CANCEL\_STAT** field of register  
3698              **INTR\_STATUS** (*Section 7.4.7*) to inform the Host that the **RSTACT** CCC and Target Reset command  
3699              sequence was cancelled (i.e., as a specific example of a transaction sequence).
  - 3700           • The Host Controller shall report the error via field **HC\_ERR\_CMD\_SEQ\_TIMEOUT\_STAT** of register  
3701              **INTR\_STATUS** (*Section 7.4.7*) to inform the Host that the Target Reset action was cancelled due to an  
3702              unresolved stall condition that became a timeout, per *Section 6.13.2*.
  - 3703           • The Host Controller shall set a flag indicating that all subsequent Command Descriptors in the  
3704              sequence would be rejected, until it receives the final Target Reset Command Descriptor with field  
3705              **RESET\_OP\_TYPE** set to the value 0x3.
  - 3706           • Any Command Descriptor that is rejected (i.e., that arrived after the I3C Bus Controller Logic was  
3707              forced to drive a STOP condition) shall cause a Response Descriptor to be generated, with error code  
3708              0x8 (**HC\_ABORTED**) in field **ERR\_STATUS**.
  - 3709           • When the Host Controller receives the final Target Reset Command Descriptor to leave the Critical  
3710              Section, it shall resume processing any subsequent Command Descriptors as normal, and then  
3711              generate a Response Descriptor with error code 0x6 (**OVL**) in field **ERR\_STATUS** to signal failure to  
3712              drive any Target Reset action due to underflow of the Command Queue/Ring.
- 3713     • If the Host Controller processes all the Command Descriptors in a sequence, without error due to  
underflow or NACK, and without cancellation or other issue, then:
  - 3714       • Once the Host Controller receives the last Transfer Command for the **RSTACT** Direct SET CCC with  
3715           **TOC**=1, the Host Controller shall drive the Target Reset Pattern after the end of that Direct SET CCC  
3716           segment, followed by a Repeated START and then a STOP.
  - 3717       • When the Host Controller receives the final Target Reset Command Descriptor to leave the Critical  
3718           Section, it shall resume processing any subsequent Command Descriptors as normal, and then generate  
3719           a Response Descriptor with error code 0x0 (**SUCCESS**).

3720     **Note:**

3721       *The Host Controller shall always generate a Response Descriptor when it receives the Target  
3722       Reset Command Descriptor to leave the Critical Section (i.e., when field **RESET\_OP\_TYPE** has a  
3723       value of 0x3), either for a successful sequence or for any sequence-related error condition during  
3724       the Critical Section.*

3725       *The Host Controller shall not generate a Response Descriptor for this Target Reset Command  
3726       Descriptor, if an individual transfer within the sequence fails with error code 0x5 (**NACK**) or 0x4  
3727       (**ADDR\_HEADER**), as Transfer Command processing would be halted by this error.*

3728     Software should follow these requirements and recommendations:

- 3729       • All Transfer Commands for the Atomic transaction should be a sequence comprised of Broadcast CCCs  
3730       and Direct SET CCCs, for the **RSTACT** CCC, in continuous framing.
  - 3731           • All Transfer Commands must be in SDR Mode, at any supported data rate. Transfer Commands in  
3732              HDR Modes are not supported.
  - 3733           • If the sequence has two or more Transfer Commands, the last such Command Descriptor must set field  
3734              **TOC** to 1'b1, and all other Command Descriptors before the last must set field **TOC** to 1'b0.
- 3735       • All such Transfer Commands in the sequence should be Immediate Data Transfer Commands per  
3736       *Section 7.1.2.1* of the I3C TCRI Specification [**MIPI06**]. Other types of Transfer Commands are not  
3737       recommended. Combo Transfer Commands shall not appear in such a sequence

- 3738 • The first Transfer Command should generally be a Broadcast CCC with a Defining Byte value of 0x00,  
3739 to instruct all Target Devices to disable Reset on seeing the Target Reset Pattern.
- 3740 • If a Broadcast CCC is used, then it should be the first Transfer Command in the sequence (after the  
3741 Target Reset Command Descriptor with field **RESET\_OP\_TYPE** set to a value of 0x2), and it should not  
3742 be sent again at any later point in the sequence.
- 3743 • The Broadcast CCC with a Defining Byte value of 0x00 is recommended. Use of other Defining Byte  
3744 values is not recommended, for most use cases of the **RSTACT** CCC with Target Reset Pattern.
- 3745 • Subsequent Transfer Commands should be Direct SET CCCs, to address individual Target Devices.

3746 The Host Controller shall generate Response Descriptors as normal for all Command Descriptors following  
3747 the initial Target Reset Command Descriptor to enter the Critical Section. The Host Controller shall also  
3748 monitor the transaction status of each Command Descriptor that uses the Direct SET form of the **RSTACT**  
3749 CCC, to ensure that the indicated Target Device ACKs its Dynamic Address. If all such Target Devices  
3750 respond properly to the **RSTACT** CCC for their respective Command Descriptor in the Atomic transaction,  
3751 then the Host Controller will know that each addressed Target Device is ready to accept the indicated Target  
3752 Reset action, and the Host Controller will drive a single Target Reset Pattern at the appropriate time, instead  
3753 of simply driving the STOP (P) condition for **TOC**=1, as with normal SDR framing (see the I3C  
3754 Specification [**MIPI02**] at *Section 5.1.11.1*).

3755 However, if any Target Device does not ACK its Dynamic Address, or fails to respond to the **RSTACT**  
3756 CCC, then the Host Controller shall **not** drive the Target Reset Pattern. Instead, the Host Controller shall  
3757 terminate the Atomic transaction and leave the Critical Section after the failed transaction, and immediately  
3758 drive the STOP (P) condition, even if the Command Descriptor for that transaction indicated continuation  
3759 of framing with field **TOC**=0. The Host Controller shall generate a Response Descriptor for the failed  
3760 transfer, with an appropriate error code in field **ERR\_STATUS** (see *Section 6.13.1*).

3761 Additionally, the Host Controller shall assert an interrupt via the **HC\_SEQ\_CANCEL\_STAT** field of register  
3762 **INTR\_STATUS** (see *Section 7.4.7*) to inform the Host of this error. Software should perform an appropriate  
3763 error recovery procedure to determine why any of the Target Devices did not respond to the **RSTACT** CCC  
3764 as part of this Atomic transaction.

3765 Software should monitor this process to ensure a successful Target Reset Action, and to understand errors  
3766 or other issues that might arise.

- 3767 • If software receives a Response Descriptor for the Target Reset Command Descriptor to leave the Critical  
3768 Section, then it knows that the Host Controller has resumed normal command processing.
- 3769 • If this Response Descriptor contains any error code other than 0x0 (**SUCCESS**), then software should  
3770 determine the appropriate action to take, based on the error code as well as the error codes in the  
3771 Response Descriptors for the individual Transfer Commands in the sequence (i.e., the Broadcast CCC  
3772 and the Direct SET CCCs).
- 3773 • If software receives an error that indicates Command/Response processing has halted due to failure i.e.,  
3774 **NACK**) of an individual transfer in the sequence, then it shall run an error recovery procedure.

### 6.15.2 Controller SDA Recovery or Bus Reset Procedures

The Host Controller shall support several new procedures added in version 1.1.1+ of the I3C Specification for recovering the I3C Bus after a Controller reset, or from a Target Device with a stuck SDA Line (see the I3C Specification [*MIPI05*] at *Section 5.1.10.2.6* and *Section 5.1.10.2.7*). The Host Controller supports these flows using Command Descriptors of type Internal Control Command (see *Section 8.4.2*) with the value of 0x5 in field **MIPI\_CMD** (see *Table 135* and *Table 140*). In this section and its subsections, such a Command Descriptor is referred to as a “Recovery Reset Command Descriptor”.

#### 6.15.2.1 Recovering from a Stuck SDA Lane

Software may issue a procedure to recover from a stuck SDA Lane by issuing a Recovery Reset Command Descriptor with the appropriate options (see *Table 140*) in order to run the appropriate I3C recovery procedures (see the I3C Specification [*MIPI02*] at *Section 5.1.10.2.6*). Field **REC\_RESET\_PROC** occupies Bits[15:12].

- If reading from an I<sup>2</sup>C Device, then software should enqueue a Recovery Reset Command Descriptor with value 0x0 in field **REC\_RESET\_PROC**.
- If reading from an I3C Device with SDR Mode, then software should enqueue a Recovery Reset Command Descriptor with value 0x1 in field **REC\_RESET\_PROC**.
- If reading from an I3C Device with HDR-DDR Mode, then software should enqueue a Recovery Reset Command Descriptor with value 0x2 in field **REC\_RESET\_PROC**.

The Recovery Reset Command Descriptor should also use an appropriate SCL speed value in field **SCL\_SPEED** (i.e., Bits[19:16]). The SCL speed value is aligned with the values supported by the various types of Command Descriptor (*Section 8.4*) as used for the **MODE** field. Generally, software should choose an SCL speed value that is supported by the particular Target Device.

#### 6.15.2.2 Forcing a STOP Condition

If an I3C Target Device fails to respond to some transfers, or if there are other transmission errors, then forcing a STOP Condition may be used as part of an error recovery procedure. Software may force a STOP Condition by issuing a Recovery Reset Command Descriptor with the appropriate options (see *Table 140*) in order to run the appropriate recovery procedure for various error conditions (see the I3C Specification [*MIPI02*] at *Section 5.1.10.2*). This procedure causes the I3C Bus Controller Logic to force a STOP after the previous command transfer; however, if both the SDA and SCL Lanes are already High, then the I3C Bus Controller Logic shall drive a START, followed by the Broadcast Address (i.e., 7'h7E) and a STOP. This ensures that a STOP is always driven, even if there was no current continuation of the frame or if there was a Bus Free Condition of any duration from a previous transfer.

Software should enqueue a Recovery Reset Command Descriptor with the value 0x4 in the **REC\_RESET\_PROC** field (i.e., Bits[15:12]), and an appropriate value in the **SCL\_SPEED** field (i.e., Bits [9:16]). The SCL speed value is aligned with the values supported by the various types of Command Descriptor (see *Section 8.4*) used for the **MODE** field. Generally, software should choose an SCL speed value that is supported by any particular non-responsive Target Devices, or by the slowest Target Device present on the I3C Bus, depending on the nature of the transmission error.

### 6.15.2.3 CE2 Error Handling for a Non-Responsive I3C Target

If an I3C Target Device does not respond to a Private read/write transaction, then the HDR Exit Pattern may be used as part of an error recovery procedure. Software may issue the HDR Exit Pattern by issuing a Recovery Reset Command Descriptor with the appropriate options (see *Table 140*) in order to run the appropriate recovery procedure (see the I3C Specification [*MIPI02*] at *Section 5.1.10.2.5*). This procedure is supported even if the Host Controller does not implement support for any HDR Modes.

Software should enqueue a Recovery Reset Command Descriptor with the value 0x5 in the **REC\_RESET\_PROC** field (i.e., Bits[15:12]), and an appropriate value in the **SCL\_SPEED** field (i.e., Bits[19:16]). The SCL speed value is aligned with the values supported by the various types of Command Descriptor (see *Section 8.4*) as used for the **MODE** field. Generally, software should choose an SCL speed value that is supported by the particular non-responsive Target Device.

### 6.15.2.4 Controller Recovery After Crash or Unexpected Reset

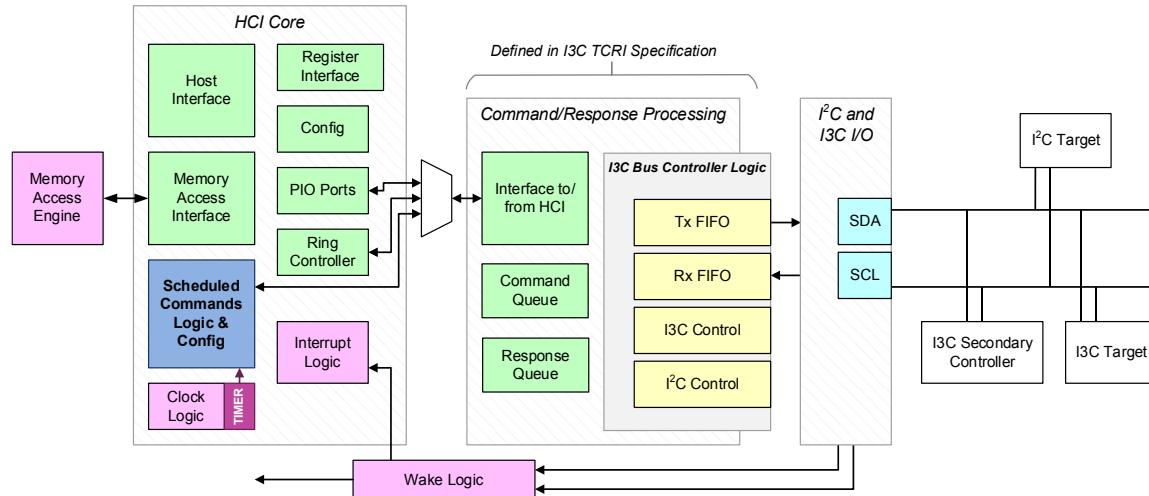
If the Controller inside the Host Controller crashes, or is reset unexpectedly, and does not know the context of the Bus, then the Controller must determine the Bus' state. Software may issue the Bus Reset procedure by issuing a Recovery Reset Command Descriptor with the appropriate options (see *Table 140*) in order to run the appropriate I3C recovery procedure (see the I3C Specification [*MIPI02*] at *Section 5.1.10.2.7*).

Software should enqueue a Recovery Reset Command Descriptor with value 0x7 in the **REC\_RESET\_PROC** field (i.e., Bits[15:12]), and an appropriate SCL speed value in the **SCL\_SPEED** field (i.e., Bits[19:16]). The SCL speed value is aligned with the values supported by the various types of Command Descriptor (see *Section 8.4*) as used for the **MODE** field. Generally, software should choose an SCL speed value that is supported by the slowest Target Device present on the I3C Bus.

## 6.16 Scheduled Commands

The Host Controller may optionally support Scheduled Commands. The Scheduled Commands capability allows for issuing certain types of Transfer Commands, according to a programmed schedule based on an internal timer. This allows Host software to offload frequent transfers to the Host Controller.

**Figure 29** shows a high-level example of a Host Controller implementation that supports the Scheduled Commands capability. This implementation is derived from the one shown in **Figure 3**, adding the Scheduled Commands logic along with its registers (which are exposed in the Scheduled Commands Extended Capability structure, **Section 7.7.8** or **Section 7.7.9**), as well as the necessary additions to the Clock Logic, such as a timer that allows the Host Controller to automatically execute Transfer Commands according to its configuration.



**Figure 29 Example Host Controller Implementation with Scheduled Commands Logic**

**Note:**

In **Figure 29**, a MUX represents the mechanism for selecting the operating context. This mechanism allows the I3C Bus Controller Logic to use either the current operating mode (i.e., PIO vs. DMA) or the Scheduled Commands logic, as and when the Clock Logic determines that Scheduled Commands should be executed (i.e., at the soonest opportunity). The MUX should only switch to use Scheduled Commands logic when the current operating mode has finished sending a sequence of Command Descriptors (i.e., should not switch in the middle of a sequence, nor when other Command Descriptors are still enqueued). When the Scheduled Commands logic has finished its execution, the MUX switches back to the current operating mode.

If the Host Controller supports the Scheduled Command capability, and if software chooses to configure it for use, then software may read and then configure the appropriate registers in the Extended Capability structure. Software must then set field **R/S**, in the specific configuration register for this Extended Capability structure, to enable automatic processing of Scheduled Commands. Software must subsequently clear the **R/S** field to disable automatic processing, in order to make configuration changes or modify entries in the Scheduled Commands structure.

**Note:**

If the Driver initiates a global Host Controller Abort operation, then all Scheduled Commands processing shall be aborted (per **Section 6.8.4**) as part of entering the **ABORTED** state. Within each Scheduled Commands Extended Capability structure, field **R/S** in register **SCHEDULE\_CONFIG** shall be cleared (i.e., shall be set to 1'b0) to stop further processing of Scheduled Commands. To continue processing, the Driver must then re-enable Scheduled Commands after clearing the

3860      *global ABORTED state, by writing 1'b1 to field R/S. See **Section 7.7.8.3**, **Section 7.7.9.3** and*  
3861      ***Section 7.7.10.3**.*

3862      The implementer may choose which Transfer Command format the Scheduled Commands logic supports:

- For generalized applications, the Scheduled Commands logic might support only Transfer Command Format 1 (which is the same format used for Command Queue/Ring(s); see **Section 8.4**).
- For specialized applications, the Scheduled Commands logic might support only Transfer Command Format 2 (which is defined in **Section 7.2.2** of the I3C TCRI Specification [**MIPI06**]), or it might support both Transfer Command Formats 1 and 2 (per **Annex A, Section A.2** of the I3C TCRI Specification).

3863      A Host Controller may support any of several different Handler Types for Scheduled Command  
3864      capabilities, either singly or in combinations (per **Table 8**). Each Handler Type has its own, unique  
3865      Extended Capability structure.

3871      **Table 8 Supported Scheduled Command Capability Handler Types**

Handler Type	Description	Limit	Sections
0	Reserved for implementer-defined capability	1 only	<b>N/A</b>
1	Simple Table (up to 16 entries)	1 only	<b>6.16.2</b> and <b>7.7.8</b>
2	Command Sequence (configurable, register-based)	Up to 8 (total across both Handler Types)	<b>6.16.3</b> and <b>7.7.9</b>
3	Schedule Buffer (configurable, DMA-based)		<b>6.16.4</b> and <b>7.7.10</b>
4 – 15	Reserved for future definition by MIPI Alliance Software WG	–	–

3872      **Note:**

3873      *Implementers may choose to support various combinations of the defined Handler Types, based on*  
3874      *the specific use case for a Host Controller. If a Host Controller supports multiple Handler Types,*  
3875      *then each shall have one or more instances of Extended Capability structures (as limited by **Table***  
3876      ***8**) with separate configuration registers in each structure. For each structure, field R/S in register*  
3877      **SCSCHEDULE\_CONFIG** *shall control whether that instance of Scheduled Commands logic is enabled*  
3878      *or disabled for processing, such that each instance can be enabled or disabled separately from*  
3879      *other instances.*

### 6.16.1 Common Aspects of Command Scheduling

The defined Handler Types provide different models for preparing Transfer Commands to be executed at regular opportunities. However, all defined Handler Types share common aspects regarding their configuration, and also share some registers in their Scheduled Commands Extended Capability structures.

The first few registers in the Scheduled Commands Extended Capability structure share similar key fields. For example:

- The first such register (i.e., **SCHEDULE\_CAPABILITIES**) defines the clock resolution of the Scheduled Commands logic, and whether the Scheduled Commands logic supports execution of the Transfer Command(s) at every **N**-th opportunity, for configurable values of **N** (where  $1 \leq N \leq 256$ ).
- The second such register (i.e., after register **SCHEDULE\_CAPABILITIES**) distinguishes the specific Handler Type for the structure, and also defines the layout of the remaining registers in the structure. The value of field **SCHED\_HANDLER** in this register will indicate one of the defined Handler Types in this I3C HCI Specification; alternately, the value 0x0 indicates an implementer-defined Handler Type (i.e., one that is not defined in this I3C HCI Specification). For implementer-defined Handler Types, the registers in such an Extended Capability structure should use the same key fields (including field **SCHED\_HANDLER**).
- The third such register (**SCHEDULE\_CONFIG**) provides control and configuration of the Handler Type instance, including whether Scheduled Commands processing is enabled for that instance.

Additional registers (i.e., after register **SCHEDULE\_CONFIG**) are specific to the Handler Type.

The Scheduled Commands logic is configurable, and attempts to execute the desired Transfer Command(s) at specific clock ticks. The tick interval is configured by the Driver (i.e., in relation to the Host Controller's clock) such that the Scheduled Commands logic can execute one or more scheduled entities in a Scheduled Commands Extended Capability structure. Each such entity can be scheduled to be executed within one or more specific tick intervals. The Scheduled Commands logic supports up to 32 unique tick intervals, and the total number of supported tick intervals comprises the tick horizon.

**Note:**

*The Scheduled Commands logic iterates over the tick intervals in ascending order, starting with tick interval 0. Implementers may determine how many tick intervals are supported per tick horizon. The minimum number of tick intervals is 8, and the recommended number of tick intervals is 32. If the implementer supports fewer tick intervals per tick horizon, then the corresponding high bits in field **SCHEDULE\_MASK** within an entity's mask register shall be read-only, for tick interval numbers that are not part of the tick horizon. For example, if only 20 tick intervals are supported, then Bits[19:0] in field **SCHEDULE\_MASK** shall be writeable, but Bits[31:20] in field **SCHEDULE\_MASK** shall be read-only and fixed to 0x0.*

The configurability of each entity varies per the Handler Type:

- **For Handler Type 1**, an entity is a specific Schedule Table Entry that contains one Transfer Command. The Driver shall configure the specific tick intervals where each entry can be executed.
  - This means that each entry's single Transfer Command will be executed independently from the others, at the configured tick intervals, per the value of field **SCHEDULE\_MASK** for that entry (i.e., in register **SCHEDULE\_TABLE\_##\_MASK**).
  - To disable a particular entry from execution, the Driver should write a value of 0x0 into field **SCHEDULE\_MASK** for that entry.
- **For Handler Type 2**, the entire Schedule Sequence is the entity. The Driver shall configure the specific tick intervals where the entire sequence can be executed (i.e., for all configured Targets).
  - This means that the sequence of Transfer Commands will be executed only at the configured tick intervals, per the value of field **SCHEDULE\_MASK** (i.e., in register **SCHEDULE\_SEQ\_MASK**). The Scheduled Commands logic will iterate over the configured Targets.

- 3925     • **For Handler Type 3**, the entire Schedule Buffer is the entity. The Driver shall configure the specific tick  
3926       intervals where the entire buffer can be executed.  
3927       • This means that the buffer of Transfer Commands will be executed only at the configured tick  
3928        intervals, per the value of field **SCHEDULE\_MASK** (i.e., in register **SCHEDULE\_BUF\_MASK**).

3929     Each instance of Scheduled Commands logic will indicate the clock resolution in field  
3930       **CLOCK\_RESOLUTION**, a read-only field in register **SCHEDULE\_CAPABILITIES** that is set by the implementer  
3931       and indicates the number of clock ticks per 1 ms. This is typically derived from the Host Controller's core  
3932       clock, and is set at implementation time. The value in field **CLOCK\_RESOLUTION** is read by the Driver, and  
3933       this gives context for how to set field **TICK\_TIME** in register **SCHEDULE\_CONFIG**. For example:

- 3934       • If the Driver reads from field **CLOCK\_RESOLUTION** and then writes that same value into field **TICK\_TIME**,  
3935        then the Scheduled Commands logic shall use a tick interval of 1 ms. (This is the default value for field  
3936        **TICK\_TIME** after a Host Controller reset.)  
3937       • If the Driver reads from field **CLOCK\_RESOLUTION**, divides that value by 2, and then writes the result into  
3938        field **TICK\_TIME**, then the Scheduled Commands logic shall use a tick interval of 500 µs (i.e., half a  
3939        millisecond).  
3940       • If the Driver reads from field **CLOCK\_RESOLUTION**, multiplies that value by 3, and then writes the result  
3941        into field **TICK\_TIME**, then the Scheduled Commands logic shall use a tick interval of 3 ms.

3942     If an instance of Scheduled Commands logic is enabled (i.e., if field **R/S** in register **SCHEDULE\_CONFIG** is  
3943       set to 1'b1), then the Host Controller shall initiate its timer, counting from tick interval 0 to tick interval 31  
3944       (or the last supported tick interval, if the Scheduled Commands logic supports fewer tick intervals) at the  
3945       configured interval (per field **TICK\_TIME**), and shall then attempt to execute the entities that are configured  
3946       to execute at each particular tick interval (per field **SCHEDULE\_MASK**). After the last tick interval in the tick  
3947       horizon, the Host Controller shall repeat the cycle starting with tick interval 0. If the entity is configured to  
3948       be executed at a particular tick interval (i.e., if the corresponding bit in field **SCHEDULE\_MASK** is set to  
3949       1'b1), then the Host Controller is able to execute that entity.

3950     It is possible for multiple entities to be scheduled to execute at the same time. This can happen within a  
3951       single instance of Scheduled Commands logic (i.e., only one Handler Type) or across multiple such  
3952       instances; however, for multiple instances, this can happen if either:

- 3953       • The instances have the same value in **TICK\_TIME**, and are configured to execute in the same tick interval;  
3954        or  
3955       • The instances have different values in **TICK\_TIME** in each instance, and the tick intervals happen to  
3956        coincide or overlap.

3957     In either of the cases above, the execution happens according to the following priority based on the Handler  
3958       Types that are supported and enabled:

- 3959       1. First: If an instance of Handler Type 1 exists and is enabled, then the Schedule Table Entries that are  
3960           enabled for this tick interval are executed in ascending order (i.e., first entry 0 if enabled, then entry 1  
3961           if enabled, etc.).
- 3962       2. Next: If one or more instances of Handler Type 2 or Handler Type 3 exist and are enabled, then the  
3963           instances execute according to the instance ID value (which is unique for each entry) in ascending  
3964           order, starting with instance 0, then instance 1, etc.
  - 3965           A. For Handler Type 2, the instance ID is in field **HT2\_INST\_ID** in register **SCHEDULE\_LAYOUT** (see  
3966              *Section 7.7.9.2*).
  - 3967           B. For Handler Type 3, the instance ID is in field **HT3\_INST\_ID** in register **SCHEDULE\_LAYOUT** (see  
3968              *Section 7.7.10.2*).

3969     The Scheduled Commands logic shall always attempt to complete execution of all scheduled entities for a  
3970       particular tick interval. Execution begins as soon as possible when the tick interval begins. However, if the  
3971       Bus Controller logic is currently busy executing a Transfer Command from the current execution context  
3972       per the operating mode (i.e., Command Queue for PIO Mode, or Command Ring for DMA Mode) or  
3973       handling an In-Band Interrupt, then execution of scheduled entities will be delayed. The Bus Controller  
3974       logic shall not interrupt a sequence of enqueued Transfer Commands from the Command Queue/Ring (i.e.,

3975 as long as field **TOC=0**) or an ongoing In-Band Interrupt, in order to start execution of any scheduled  
3976 entities.

3977 Assuming that the Scheduled Commands logic is able to start execution of scheduled entities for a given  
3978 tick interval: if execution of all scheduled entities finishes before the next tick interval starts, and if there  
3979 are Transfer Commands that are available for execution in the current execution context (i.e., the Command  
3980 Queue/Ring), then the Host Controller shall move to execute the enqueued Transfer Commands in the  
3981 current execution context.

3982 If a new tick interval starts before the Scheduled Commands logic is able to finish execution of all  
3983 scheduled entities for the previous tick interval, then the Scheduled Commands logic must skip that new  
3984 tick interval and generate an interrupt to the Host, to report the missed tick interval. This can occur with  
3985 any or all of the following scenarios:

- 3986 • Scheduling a large number of Transfer Commands (i.e., across multiple entities) that cannot all execute  
3987 within a short tick interval
- 3988 • Scheduling long Transfer Commands (i.e., many data bytes transferred to/from a Target)
- 3989 • Scheduling slow Transfer Commands (i.e., reduced data transfer rate on the I3C Bus)
- 3990 • Clock stalling due to overflow or underflow of the data buffer (for PIO Mode only)
- 3991 • IBI Queue/Ring overflowing, because Host does not drain the IBI Queue/Ring in time to prevent a stall
- 3992 • Longer sequences of enqueued Transfer Commands in a command sequence (i.e., with field **TOC=0**) in  
3993 the current execution context

3994 The following example shows how the Host Controller determines that it must skip a new tick interval:

- 3995 1. One or more instances of Scheduled Commands logic are configured with **TICK\_TIME** of 5 ms.
- 3996 2. At time **T + 0**, tick interval 1 begins. The Bus Controller logic is not busy, so the Scheduled Commands  
3997 logic begins executing the scheduled entities for tick interval 1.
- 3998 3. The Bus Controller logic takes 3 ms to execute these entities for tick interval 1, and then the Bus goes  
3999 idle.
- 4000 4. At time **T + 5**, tick interval 2 begins, but the Bus Controller logic is busy processing an In-Band  
4001 Interrupt from an I3C Target and/or a sequence of Transfer Commands in the current execution  
4002 context.
- 4003 5. After 3 ms (time **T + 8**), the Bus Controller logic finishes these actions and begins executing the  
4004 scheduled entities for tick interval 2. The Host Controller does not assert an interrupt, since tick  
4005 interval 2 was not missed.
- 4006 6. The Bus Controller logic takes 4 ms to fully execute all such scheduled entities for tick interval 2,  
4007 before the I3C Bus becomes idle.
- 4008 7. At time **T + 10**, tick interval 3 begins. The Scheduled Commands logic sees the start of this new tick  
4009 interval, but is unable to execute any scheduled entities for tick interval 3 (i.e., because the I3C Bus  
4010 Controller logic was still busy processing scheduled entities for tick interval 2).
- 4011 8. The Host Controller asserts an interrupt that indicates that a tick interval was missed.
- 4012 9. The Bus Controller logic remains busy until time **T + 12** and then goes idle.
- 4013 10. At time **T + 15**, tick interval 4 begins. The Scheduled Command logic sees the start of this new tick  
4014 interval, and is able to execute the scheduled entities.

4015 **Note:**

4016 *The example above is not an exhaustive definition of when a tick interval could be skipped. In this  
4017 particular example, it would have been possible for the scheduled entities for tick interval 2 to fully  
4018 execute within the 5 ms time window, if the I3C Bus was idle at the start of the tick interval.  
4019 However, the ongoing I3C Bus activity at the start of tick interval 2 delayed the execution of the  
4020 scheduled entities.*

4021 *For Handler Type 3 only: if the Schedule Buffer could not be executed because field **EXECUTED** in  
4022 register **SCHEDULE\_BUF\_CONFIG** (see **Section 7.7.10.5**) was set to 1'b1, this does not mean that a  
4023 tick interval was missed.*

4024     *The Host Controller shall report interrupts for missed tick intervals, by asserting field*  
4025     **SCHED\_CMD\_MISSED\_TICK\_STAT** *in register* **INTR\_STATUS** *(see* **Section 7.4.7***). Implementers may*  
4026     *also choose to support the debug register* **SCHED\_CMDS\_DEBUG** *for Scheduled Commands logic*  
4027     *(see* **Section 7.7.7.5***). If supported, then this debug register shall indicate the specific ID of the*  
4028     *missed tick interval, for such situations where a tick interval is missed.*

### 6.16.2 Handler Type 1: Simple Table

In order to support the Scheduled Commands capability with Handler Type 1, the Host Controller shall expose a Schedule Table with up to 16 unique entries. Each entry in the Schedule Table may be programmed to be executed on specific schedule ticks across the schedule horizon. The Schedule Table shall be exposed as registers in an Extended Capability of type Scheduled Commands, per [Section 7.7.8](#).

The Schedule Table comprises an array of Schedule Table Entries, with each entry containing registers to hold the schedule tick configuration, as well as additional registers to hold one Command Descriptor. Each such Command Descriptor shall be a valid Transfer Command describing either a Read-Type transfer with certain restrictions (such as a Regular Data Transfer or a Combo Transfer), or a Write-Type Immediate Data Transfer (i.e., a write with Immediate Data Bytes in the Command Descriptor).

- **For Read-Type Transfers:** The data for a specific transfer that is being read from a Target Device shall be provided on either the IBI Port (for PIO Mode) or an IBI Pair within a Ring Bundle (for DMA Mode).

If the Host Controller is operating in DMA Mode, then Ring Bundle 0 will always be used for such notifications.

- **For Write-Type Transfers:** Write data shall be provided in the Command Descriptor (i.e., only Immediate Writes are allowed for Scheduled Commands).

The Host Controller shall execute each Scheduled Command either exactly at its programmed time, or at the earliest opportunity after the programmed time. The execution of Scheduled Commands for a given schedule tick (i.e., the scheduled event for a command execution) shall not be interleaved with regular Command Queue servicing (for PIO Mode) or Command Ring servicing for any Ring Bundle that is enabled and running (for DMA Mode).

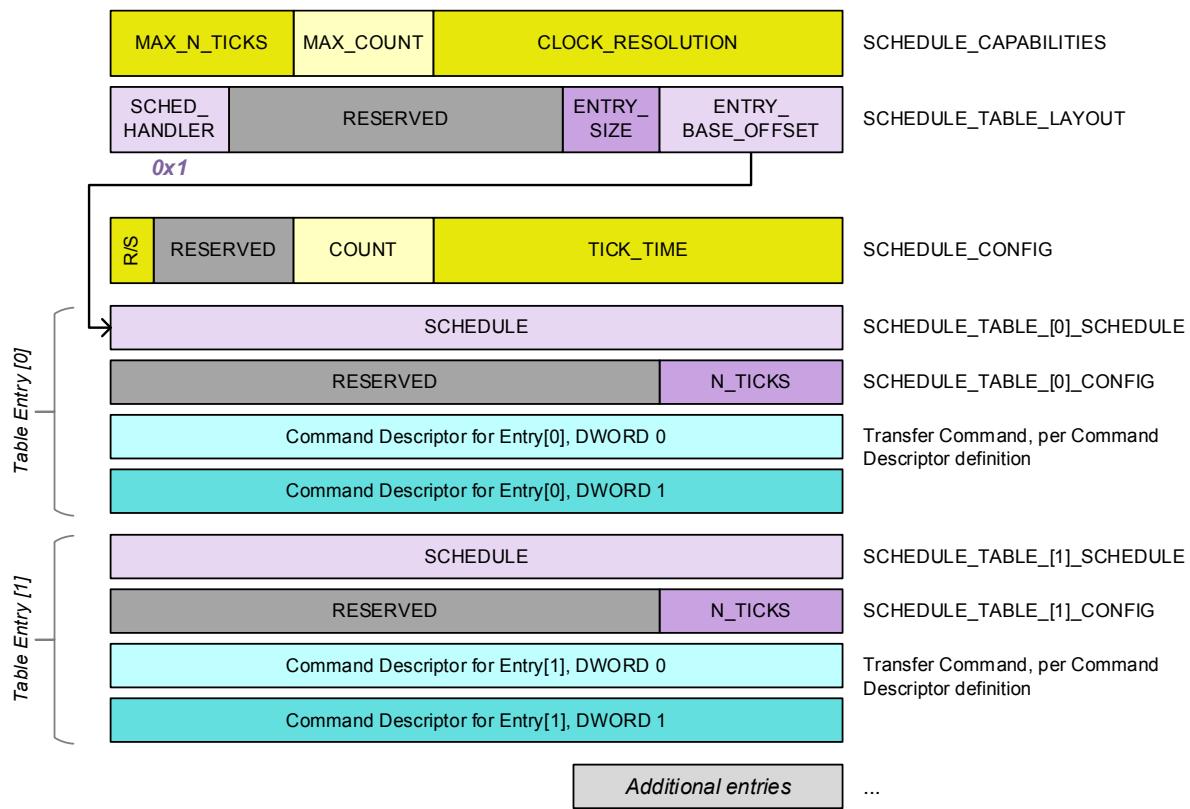
The schedule tick for each table entry is programmable, and the schedule horizon is set to 32 schedule ticks (unless limited by the implementer). Optionally, the Host Controller may support execution on every  $N$ -th opportunity (where  $N \leq 256$ ) to allow additional flexibility. For a 1 ms tick, this allows for an horizon of  $(32 \times 256 =) 8160$  ms.

The specific registers defined in subsections of [Section 7.7.8](#) are arranged into an Extended Capability structure, which may exist within the chain of Extended Capability Structures. An instance of this Extended Capability structure includes the initial registers (i.e., registers **SCHEDULE\_CAPABILITIES**, **SCHEDULE\_TABLE\_LAYOUT**, and **SCHEDULE\_CONFIG**), which serve as a descriptor for the structure, followed by an array of Schedule Table Entries. Each Schedule Table Entry consists of registers to hold the schedule configuration and the Command Descriptor.

**Note:**

An implementer may define up to one Extended Capability structure of this type. Additionally, each Schedule Table entry is limited to one Command Descriptor (i.e., only one Transfer Command). For applications that require longer sequences of Transfer Commands, the implementer must implement support for either Handler Type 2 (see [Section 6.16.3](#)) and/or Handler Type 3 (see [Section 6.16.4](#)).

4065  
4066 **Figure 30** shows the generic layout of the registers within the Extended Capability structure for Handler Type 1.



**Figure 30 Register Layout for Scheduled Commands Extended Capability Structure (Handler Type 1)**

4069     **Section 7.7.8** contains detailed examples of the register fields, as well as several sample configurations  
4070     based on the possible arrangements of the registers in this Extended Capability structure.

### 6.16.3 Handler Type 2: Command Sequence

In order to support the Scheduled Commands capability with Handler Type 2, the Host Controller shall expose a Schedule Sequence register set, to hold a single sequence of Command Descriptors, that can be executed for one or multiple unique Target Devices. The Schedule Sequence entry may hold up to 8 Command Descriptors, as well as additional registers to select the unique Target Devices for which the sequence of Command Descriptors will be executed. Execution of the sequence may be programmed to be executed on specific schedule ticks across the schedule horizon. The Schedule Sequence registers shall be exposed in an Extended Capability of type Scheduled Commands per **Section 7.7.9**.

Each Command Descriptor in the sequence shall be a valid Transfer Command that describes either a Read-Type transfer with certain restrictions (such as a Regular Data Transfer or a Combo Transfer), or a Write-Type Immediate Data Transfer (i.e., a write with Immediate Data Bytes in the Command Descriptor).

- **For Read-Type Transfers:** The data for a specific transfer that is being read from a Target Device shall be provided on either the IBI Port (for PIO Mode) or an IBI Ring Pair within a Ring Bundle (for DMA Mode).

- If the Host Controller is operating in DMA Mode, then the Driver shall choose which Ring Bundle will be used for such notifications.

- **For Write-Type Transfers:** Write data shall be provided in the Command Descriptor (i.e., only Immediate Writes are allowed for Scheduled Commands).

The Host Controller shall execute the sequence of Command Descriptors either exactly at its programmed time, or at the earliest opportunity after the programmed time. The execution of Scheduled Commands for a given schedule tick (i.e., the scheduled event for a command execution) shall not be interleaved with regular Command Queue servicing (for PIO Mode) or Command Ring servicing for any Ring Bundle that is enabled and running (for DMA Mode). If the configuration registers indicate multiple Targets, then the sequence shall be executed multiple times in succession, and the Host Controller shall iterate the sequence over each Target until all Targets have been processed.

The schedule tick for the sequence is programmable, and the schedule horizon is set to 32 schedule ticks (unless limited by the implementer). Optionally, the Host Controller may support execution on every **N**-th opportunity (where  $N \leq 256$ ) to allow additional flexibility. For a 1 ms tick, this allows for an horizon of  $(32 \times 256 =) 8160$  ms.

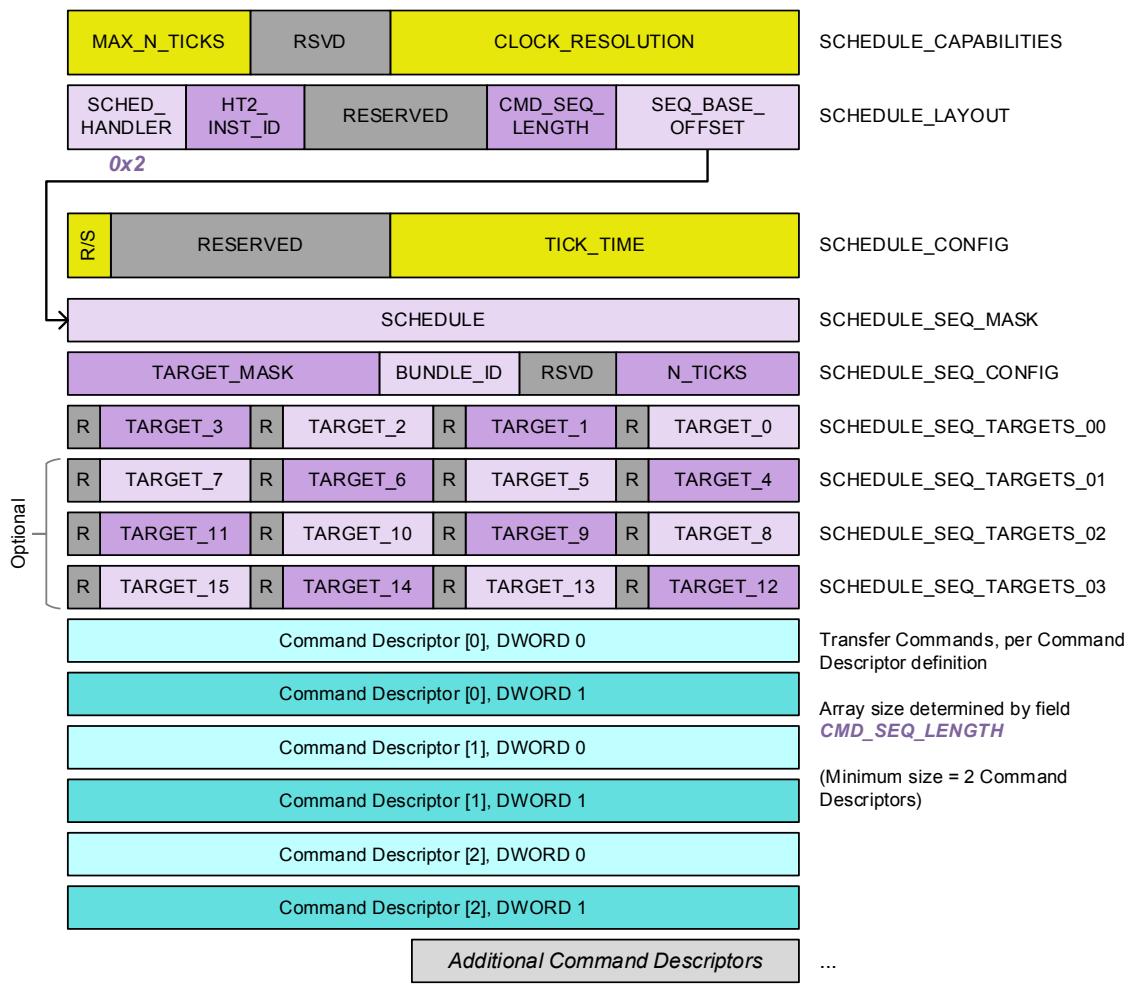
The specific registers defined in subsections of **Section 7.7.9** are arranged into an Extended Capability structure, which may exist within the chain of Extended Capability Structures. Each instance of this Extended Capability structure includes the initial registers (i.e., registers **SCHEDULE\_CAPABILITIES**, **SCHEDULE\_LAYOUT**, and **SCHEDULE\_CONFIG**), which serve as a descriptor for the structure, followed by arrays of registers for the indicated Target Devices as well as the sequence of Command Descriptors. The configuration registers are common to all indicated Targets.

**Note:**

*Within each Extended Capability structure, an implementer may determine how many Command Descriptors will be supported for the command sequence, and the **SCHEDULE\_LAYOUT** register shall indicate the size of the array. If any of the Command Descriptor registers in the array are not needed (i.e., the command sequence is shorter than the maximum number of Command Descriptors that could be used) then the Driver shall write a value of 0x7 to field **CMD\_ATTR** for any unused Command Descriptors, so that the Host Controller knows when to end the sequence.*

*An implementer may define one or multiple Extended Capability structures of this type, per the use case, in order to store multiple command sequences that are executed for unique sets of Targets.*

4114  
4115 **Figure 31** shows the generic layout of the registers within the Extended Capability structure for Handler Type 2.



4116  
4117 **Figure 31 Register Layout for Scheduled Commands Extended Capability Structure (Handler Type 2)**

4118 **Section 7.7.9** contains detailed examples of the register fields, as well as several sample configurations based on the possible arrangements of the registers in this Extended Capability structure.

#### 6.16.4 Handler Type 3: Schedule Buffer

In order to support the Scheduled Commands capability with Handler Type 3, the Host Controller shall expose a Schedule Buffer register set to describe a Schedule Buffer that can contain one or more Transfer Descriptors in Host system memory.

**Note:**

*Handler Type 3 requires the Host Controller to implement support for DMA to access the Host's system memory. Many aspects of Handler Type 3 are similar to the execution of Transfer Commands in a Command/Response Ring Pair for DMA Mode (see Section 6.6.2). However, DMA Mode is not required to be supported if Handler Type 3 is supported. If the Host Controller supports DMA Mode, then Handler Type 3 is the recommended option for Scheduled Commands capabilities.*

The Schedule Buffer is defined in Host system memory, and is an array of up to 255 Transfer Descriptors as defined in Section 8.3. Each Transfer Descriptor shall contain a Command Descriptor. The array of Transfer Descriptors shall be executed in sequence, and may be programmed to be executed on specific schedule ticks across the schedule horizon. The Schedule Buffer registers shall be exposed in an Extended Capability of type Scheduled Commands, per Section 7.7.10.

**Note:**

*The Driver shall allocate sufficient Host system memory to store the Schedule Buffer, and this memory allocation must be large enough to hold enough Transfer Commands, based on the value written into field TRANSFER\_COUNT in register SCHEDULE\_BUF\_CONFIG (see Section 7.7.10.5).*

Each such Command Descriptor in this array of Transfer Descriptors shall be a valid Transfer Command that describes either a Read-Type transfer with certain restrictions (such as a Regular Data Transfer or a Combo Transfer), or any Write-Type Transfer.

- **For Read-Type Transfers:** The Host Controller can be configured to provide read data for a specific transfer (i.e., data being read from a Target Device) on either the IBI Queue (for PIO Mode), an IBI Ring Pair within a Ring Bundle (for DMA Mode), or the Data Buffer Descriptor within the specific Transfer Descriptor.

- If the Host Controller is operating in DMA Mode, then the Driver shall choose which Ring Bundle will be used for such notifications.
- Using the Data Buffer Descriptor for read data requires the Host to pre-allocate sufficient system memory for each expected Read-Type transfer, and then populate the Data Buffer Descriptor with a valid address for receiving that read data.

- **For Write-Type Transfers:** Write data may be provided in either the Command Descriptor (i.e., an Immediate Write) or within the pre-allocated Host system memory described by the Data Buffer Descriptor within each Transfer Descriptor.

The Host Controller shall execute the sequence of Command Descriptors either exactly at its programmed time, or at the earliest opportunity after the programmed time. The execution of Scheduled Commands for a given schedule tick (i.e., the scheduled event for a command execution) shall not be interleaved with regular Command Queue servicing (for PIO Mode) or Command Ring servicing for any Ring Bundle that is enabled and running (for DMA Mode). The Host Controller shall execute the sequence of Command Descriptors in sequential order until it either reaches the end of the Schedule Buffer, or else sees a Command Descriptor that denotes the end of the sequence.

The schedule tick for the sequence is programmable, and the schedule horizon is set to 32 schedule ticks (unless limited by the implementer). Optionally, the Host Controller may support execution on every **N**-th opportunity (where **N** ≤ 256) to allow additional flexibility. For a 1 ms tick, this allows for an horizon of (32 × 256 =) 8160 ms.

The specific registers defined in subsections of Section 7.7.10 are arranged into an Extended Capability structure, which may exist within the chain of Extended Capability Structures. Each instance of this Extended Capability structure includes the initial registers (i.e., registers SCHEDULE\_CAPABILITIES,

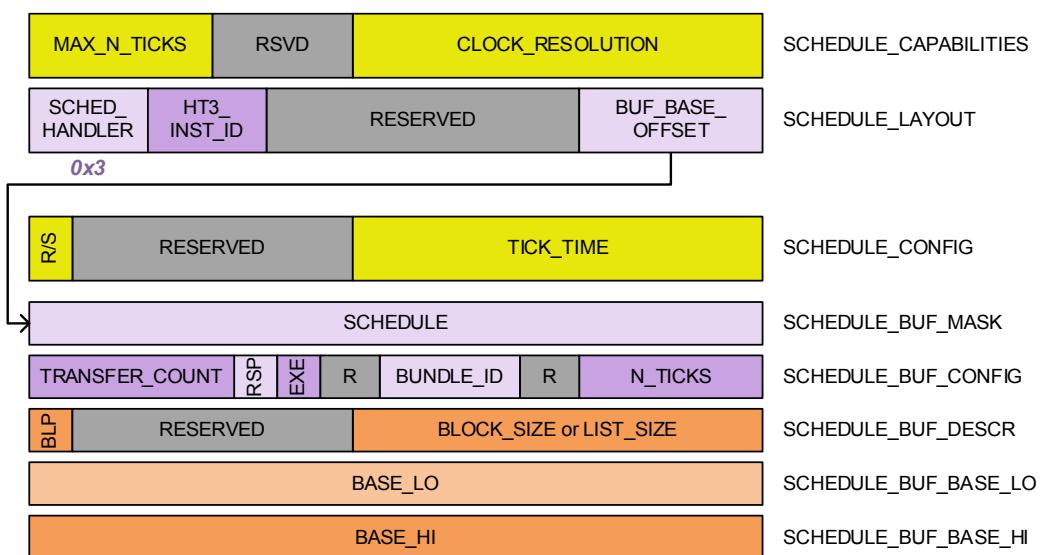
4168     **SCHEDULE\_LAYOUT**, and **SCHEDULE\_CONFIG**), which serve as a descriptor for the structure, followed by  
4169     other registers that describe and point to the Schedule Buffer.

4170     **Note:**

4171       *Within the allocated Schedule Buffer, an implementer may determine how many Transfer  
4172       Descriptors will be executed, up to the maximum limit of 255. If the number of Transfer Descriptors  
4173       is short (i.e., is less than the amount of system memory allocated by the Host), then the Driver shall  
4174       write a value of 0x7 to field **CMD\_ATTR** for any unused Command Descriptors, so that the Host  
4175       Controller knows when to end the sequence.*

4176       *An implementer may define one or multiple Extended Capability structures of this type, per the use  
4177       case, in order to describe multiple Schedule Buffers that are executed for unique sets of Targets.*

4178     Figure 32 shows the generic layout of the registers within the Extended Capability structure for Handler  
4179     Type 3.



4180     **Figure 32 Register Layout for Scheduled Commands Extended Capability Structure  
(Handler Type 3)**

4181     Section 7.7.10 contains detailed examples of the register fields, as well as several sample configurations  
4182     based on the possible arrangements of the registers in this Extended Capability structure.

## 6.17 Standby Controller Mode

The Host Controller may optionally support Standby Controller mode, which includes Secondary Controller Logic and support for transitioning between Standby Controller mode (i.e., playing the Target role but also possessing Secondary Controller capabilities) and Active Controller mode (i.e., playing the Active Controller role).

While operating in Standby Controller mode, the Host Controller shall support handling CCCs (**Section 6.17.3.1**) as though it were a limited-function I3C Target Device with additional I3C Secondary Controller capabilities.

### Minimum Requirements for Standby Controller Mode

In order to support Standby Controller mode, the following capabilities, attributes, and logic are required (these are in addition to those required to support Active Controller mode):

- Standby Controller Mode Extended Capability structure, with Extended Capability ID 0x12 (**Section 7.7.11**)
  - Support for the mandatory registers in this Extended Capability structure
- Minimal Secondary Controller Logic, including:
  - Minimum Bus Management Procedures, as defined by the I3C Specification at **Section 5.1.7.3.2 [MIPI02]**
  - Support for the Standby Controller Interrupt logic, which allows additional interrupts to be routed to the Host via the System Bus
  - Support for autonomous response to standard CCCs that are required for the role of Secondary Controller on an I3C Bus (**Section 6.17.3.1.1**)
- Controller Role FSM, including:
  - In Active Controller mode, sufficient checking of pre-Handoff readiness condition to ensure that all prerequisites for passing the Controller Role have been met, and that no other Command Descriptors would interfere with the Handoff process
  - Special command support for the sub-command that sends the **GETACCCR** CCC and initiates the Controller-to-Controller Handoff Procedure using the special Command Descriptor format, i.e., an Internal Control Command Type with field **MIPI\_CMD** having a value of 0x7 (**Section 8.4.2.7**)
  - Support for transitioning from Active Controller mode to Standby Controller mode, where the Bus Controller Logic becomes idle after successfully passing the Controller Role to another Controller-capable Device (i.e., a Secondary Controller) and Secondary Controller Logic becomes active
  - Sufficient checking of pre-handoff acceptance condition in Standby Controller mode, to ensure that all necessary steps in receiving the most current Bus configuration from the Active Controller have been performed and have been handled by Host software (see **Section 6.17.4**)
  - Support for transitioning from Standby Controller mode to Active Controller mode, where the Secondary Controller Logic goes idle after successfully accepting the Controller Role from the Active Controller, and the Bus Controller Logic becomes active
  - Support for Error Type CE3 checking, to ensure that the new Active Controller successfully asserts its ownership of the Controller Role after the Controller-to-Controller Handoff Procedure

#### Note:

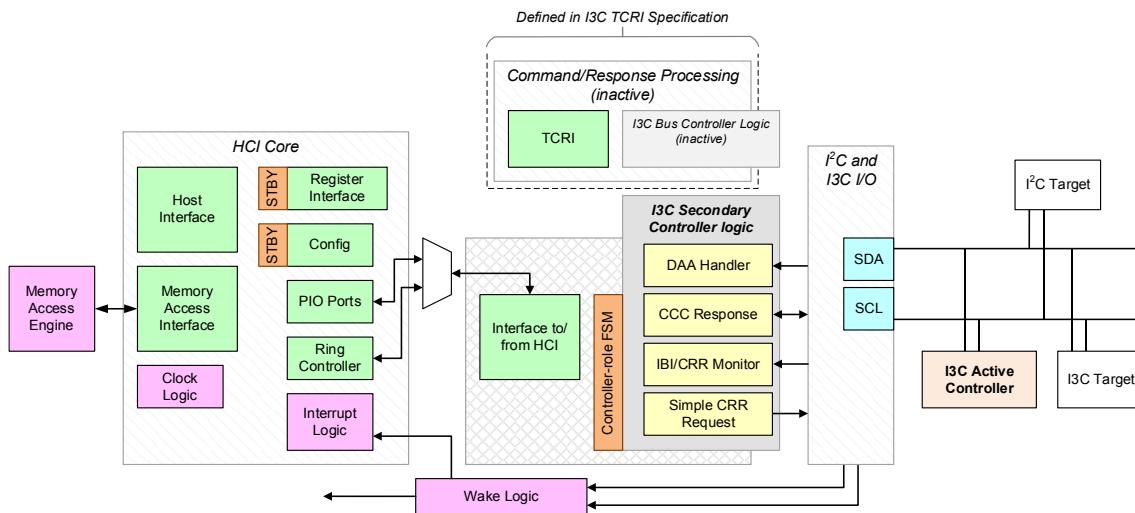
A Host Controller that only implements the minimum requirements specified above is also called a Minimal Standby Controller in this I3C HCI Specification. Such minimum requirements allow the Host Controller to pass the Controller Role to a Secondary Controller, and to receive the Controller Role again.

4226 **Optional Standby Controller Mode Capabilities**

4227 In addition to the minimum requirements, implementers may also choose to implement additional  
4228 capabilities, attributes, and logic, in order to provide additional behavior or to enable use cases that depend  
4229 on other actions required of the Host Controller while it is operating as a Standby Controller (i.e., while it is  
4230 in the Secondary Controller role).

- 4231 • Support for Initialization as a Secondary Controller on the I3C Bus:
  - 4232 • Standby Controller mode Bus connection logic, which allows the Host Controller to connect to the I3C  
4233 Bus as an I3C Target with Secondary Controller capabilities;
  - 4234 • Support for receiving a Dynamic Address from the Active Controller, using at least one supported  
4235 method of Dynamic Address Assignment (i.e., any of the defined CCCs)
    - 4236 • If Dynamic Address Assignment with **ENTDAA** is supported, then the Hot-Join Request shall also be  
4237 generated, as and when the Driver indicates that the Host Controller should Join the I3C Bus
  - 4238 • Support for receiving Bus Configuration changes from the Active Controller (i.e., via the **DEFTGTS**  
4239 and **DEFGRPA** CCCs)
- 4240 • Additional Secondary Controller Logic, including:
  - 4241 • Additional Secondary Controller functions, such as Bus Configuration Procedures, as defined at  
*Section 5.1.7.3.3* of the I3C Specification [**MIPI02**]
  - 4242 • Any other optional Secondary Controller capabilities deemed necessary by the implementer, listed in  
*Section 5.1.7.3* (and relevant sub-sections) of the I3C Specification [**MIPI02**] at *Section 5.1.7.3*
  - 4243 • Support for autonomous response to additional CCCs. This may also include CCCs supported by  
4244 extensions to Secondary Controller Logic per *Section 6.17.3.1.2*. The implementer may determine  
4245 whether the Secondary Controller Logic supports responses for any additional CCCs, and if so, which  
4246 additional CCCs will be supported.
- 4247 • Other optional capabilities and attributes, as defined by the implementer.

4250  
 4251 **Figure 33** shows a high-level example of a Host Controller implementation that supports Standby  
 4252 Controller mode. This implementation is derived from the implementation shown in **Figure 3**, adding the  
 4253 Secondary Controller Logic, additional configuration via Host Controller registers for Standby Controller  
 4254 mode (exposed in the Standby Controller Extended Capability structure, per **Section 7.7.11**), and the  
 4255 Controller Role FSM which manages the transitions between Active Controller mode and Standby  
 4256 Controller mode. This figure also illustrates such an implementation that is currently operating in Standby  
 Controller mode, with support for optional Secondary Controller capabilities.



**Figure 33 Example Host Controller Implementation with Secondary Controller Logic**

4258 If the Host Controller is configured as a Primary Controller on the I3C Bus, then it shall start operations in  
 4259 Active Controller mode, and it shall be responsible for initializing the I3C Bus as the Primary Controller  
 4260 (see the I3C Specification [**MIPI02**] at **Section 5.1.7**).  
 4261

If the Host Controller is configured as a Secondary Controller on the I3C Bus, then it shall start operations in Standby Controller mode, and shall act as a Target on the I3C Bus, and shall report that it is a Controller-capable Device (i.e., that it has Secondary Controller capabilities). The Device may request the Controller Role from the Active Controller, and if it successfully receives the Controller Role from an Active Controller (e.g., the Primary Controller), then it will transition to Active Controller mode (see the I3C Specification [**MIPI02**] at **Section 5.1.6.3**). While operating in Standby Controller mode, the functionality and logic within the Host Controller that is active and that handles configuration and transactions as a limited-capability I3C Target on the I3C Bus (i.e., in the Target role including Secondary Controller capabilities) is called Secondary Controller Logic, to distinguish it from the Bus Controller Logic (which is active and handles configuration and transactions as an I3C Controller in Active Controller mode, fulfilling the Active Controller role).

#### **Additional Active Controller Requirements**

If the Host Controller supports Standby Controller mode and includes Secondary Controller logic, then the Bus Controller Logic shall also support the following capabilities:

- If the Host Controller is a Secondary Controller and subsequently becomes the Active Controller (as described above), then it may subsequently pass the Controller Role to another Controller-capable Device, using the **GETACCCR** CCC (per **Section 6.17.2**).
- If the Host Controller (as Active Controller) passes the Controller Role to another Secondary Controller, but that Secondary Controller does not respond in time or does not assert its Controller Role status (i.e., according to Error Type CE3), then the Host Controller shall reclaim the Controller Role and become Active Controller again.

4282 **Initialization and FSM**

4283 The transitions between Active Controller mode and Standby Controller mode are handled by the  
4284 Controller Role FSM. Field **AC\_CURRENT\_OWN** in register **PRESENT\_STATE** (see [Section 7.4.6](#)) indicates  
4285 whether or not the Bus Controller Logic is active, i.e., whether or not the Host Controller is the Active  
4286 Controller and is responsible for driving the SCL (clock) line.

4287 **Note:**

4288 *The operation and initial state of the Controller Role FSM will depend on whether the Host  
4289 Controller also supports the optional Dead Bus Recovery Mechanism (see [Section 6.18](#),  
4290 [Section 7.7.6](#), and [Section 8.4.2.8](#)). If this optional mechanism is supported, then the Host may  
4291 choose whether to select the role on startup (i.e., based on Driver configuration), or else engage  
4292 the Dead Bus Recovery mechanism to attempt to detect whether the I3C Bus already has an Active  
4293 Controller. If the Host uses the Dead Bus Recovery Mechanism, then fields **ACR\_FSM\_OP\_SELECT**  
4294 and **AC\_CURRENT\_OWN** are automatically set based on the operation of the Dead Bus Recovery  
4295 Mechanism, and the Driver should not write to these fields.*

4296 *However, if the Host Controller does not also support the Dead Bus Recovery Mechanism, then the  
4297 Host shall choose to select the role on startup (i.e., based on Driver configuration) according to the  
4298 initialization flow in this section. The remainder of this section generally assumes that the Dead Bus  
4299 Recovery Mechanism is either not supported, or that the Driver has chosen not to use the Dead  
4300 Bus Recovery Mechanism.*

4301 In Standby Controller mode, field **ACR\_FSM\_OP\_SELECT** in register **STBY\_CR\_CONTROL** ([Section 7.7.11.1](#))  
4302 is used to enable readiness to accept the Controller Role and to enable the Controller-to-Controller Handoff  
4303 Procedure, thereby becoming the Active Controller; or to cancel such readiness and remain a Secondary  
4304 Controller in Standby Controller mode. In Active Controller mode field **ACR\_FSM\_OP\_SELECT** is also used,  
4305 but in this case to allow software to enqueue the special command to pass the Controller Role to another  
4306 Controller-capable Device (i.e., a Secondary Controller) on the I3C Bus, initiating the Controller-to-  
4307 Controller Handoff Procedure per [Section 5.1.7.2](#) of the I3C Specification [[MIPI02](#)].

4308 The Controller Role FSM allows flexibility for an implementer to customize a Host Controller for various  
4309 use cases:

- 4310 • [Figure 34](#) shows a variant of the Controller Role FSM for a Host Controller that only supports  
4311 initialization in Active Controller mode (i.e., as the Primary Controller of the I3C Bus) and supports the  
4312 minimal Standby Controller mode capabilities. In this variant, the Host Controller would always be  
4313 primed to accept the Controller Role after passing the Controller Role to another Secondary Controller  
4314 Device.
- 4315 • [Figure 35](#) shows a variant of the Controller Role FSM for a Host Controller that supports initialization in  
4316 either Active Controller mode (i.e., as the Primary Controller of the I3C Bus) or Standby Controller mode  
4317 (i.e., as a Secondary Controller with optional features and capabilities).

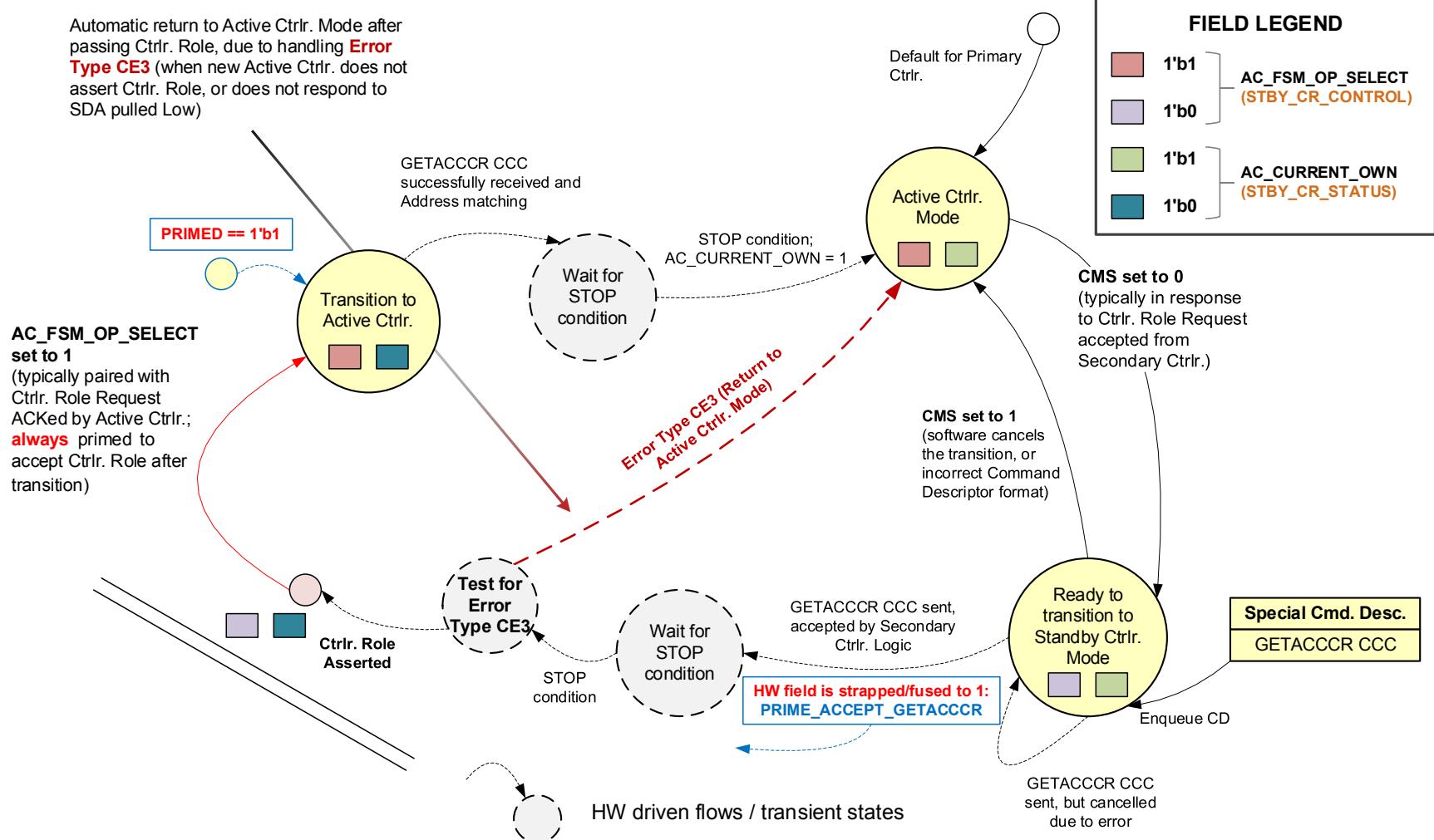
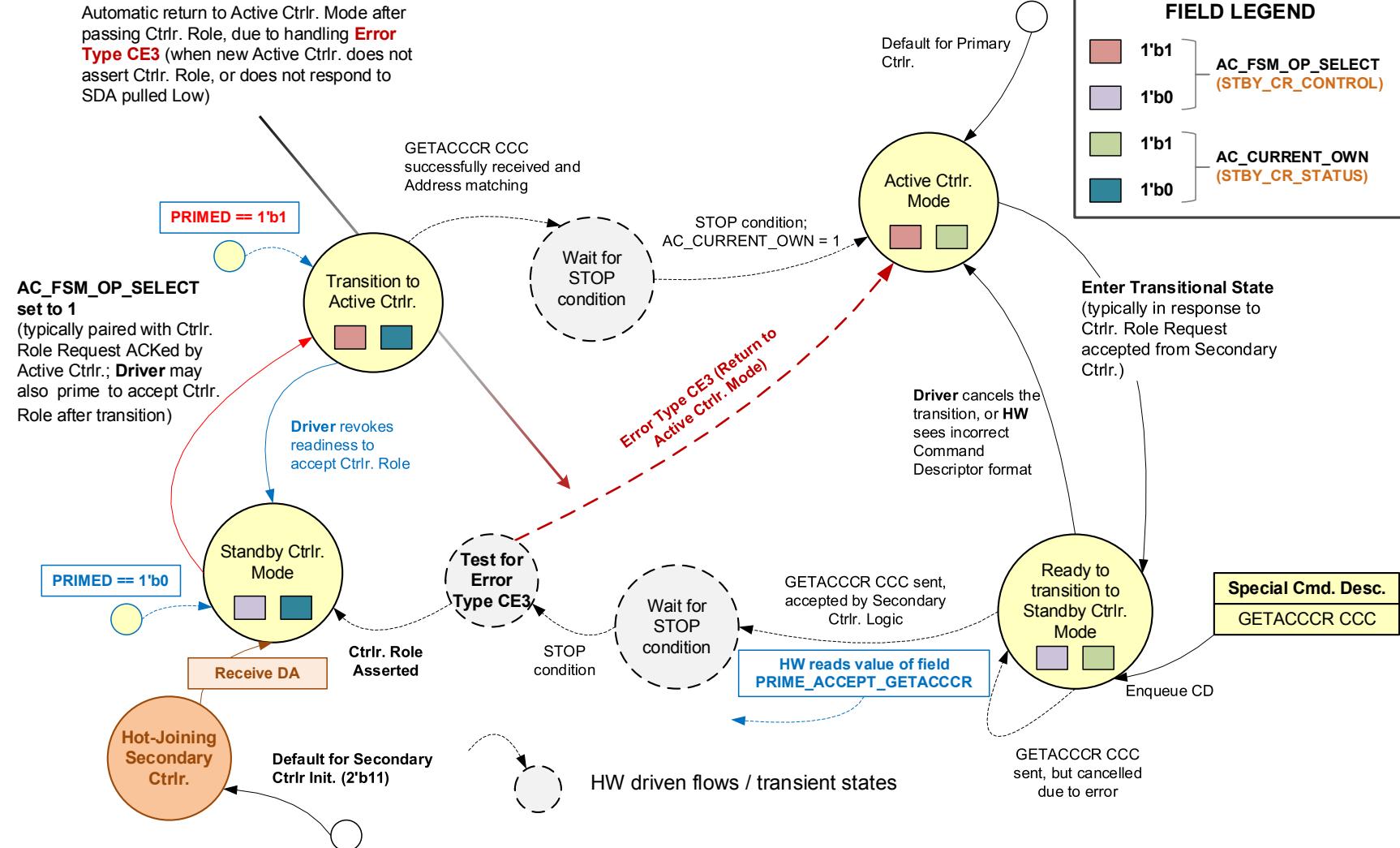


Figure 34 Controller Role FSM with Minimal Secondary Controller Capabilities



### 6.17.1 Configuration and Modes of Operation

The Driver shall configure the Host Controller, along with its I3C Bus Controller Logic and I3C Secondary Controller Logic, in the appropriate manner and order. The order of the steps to configure a Host Controller must account for the supported capabilities, and should proceed according to the desired flow and mode of operation.

#### Configuration of I3C Secondary Controller Logic

The Driver shall determine which of the following configuration flow options to use to set the Host Controller's initial role on the I3C Bus and its initial mode of operation.

- **Option 1: I3C Primary Controller Only** (i.e., Active Controller mode) **without support for Controller-to-Controller Handoff Procedure**:

- A. The Driver initializes only the Host Controller's I3C Bus Controller Logic.
- B. The Driver then enables only the Host Controller's I3C Bus Controller Logic, by writing a value of 1'b1 to field **BUS\_ENABLE** in register **HC\_CONTROL** (*Section 7.4.2*).
- C. The Host Controller begins operation in Active Controller mode.
- D. Transitions from Active Controller mode to Standby Controller mode are not possible.

- **Option 2: I3C Primary Controller** (i.e., Active Controller mode) **with support for Controller-to-Controller Handoff Procedure**:

- A. The Driver initializes both the Host Controller's I3C Bus Controller Logic and its I3C Secondary Controller Logic (per *Section 6.17.1.2*).
- B. The Driver then enables the I3C Bus Controller Logic (see **Option 1**, step B above); sets the initial mode and role by writing a value of 1'b1 to field **ACR\_FSM\_OP\_SELECT** in register **STBY\_CR\_CONTROL** (*Section 7.7.11.1*); and then enables the I3C Secondary Controller Logic by writing a value of 2'b01 to field **STBY\_CR\_ENABLE\_INIT** in register **STBY\_CR\_CONTROL**.
- C. The Host Controller begins operation in Active Controller mode. The Secondary Controller Logic remains in a standby state because the Bus Controller Logic is active (i.e., as the Primary Controller).
- D. Transitions from Active Controller mode to Standby Controller mode are possible, using the pre-Handoff steps followed by the special Command Descriptor (per *Section 6.17.2*).

- **Option 3: I3C Secondary Controller** (i.e., Standby Controller mode) **with support for accepting the Controller Role from the Active Controller**:

- A. The Driver initializes both the Host Controller's I3C Bus Controller Logic and its I3C Secondary Controller Logic (per *Section 6.17.1.3*).
- B. The then Driver sets the initial mode and role by writing a value of 1'b0 to field **ACR\_FSM\_OP\_SELECT** in register **STBY\_CR\_CONTROL** (*Section 7.7.11.1*); enables the I3C Secondary Controller Logic by writing a value of either 2'b10 or 2'b11 to field **STBY\_CR\_ENABLE\_INIT** in register **STBY\_CR\_CONTROL** (*Section 7.7.11.1*); and then enables the I3C Bus Controller Logic (see **Option 1**, step B above).
- C. The Host Controller begins operation in Standby Controller mode. The Bus Controller Logic remains in a standby state, and the Secondary Controller Logic becomes ready to respond to Dynamic Address Assignment (see *Section 6.17.1.3*).
- D. Transitions from Standby Controller mode to Active Controller mode are possible, provided that the Host Controller has received a Dynamic Address, and that the Driver has primed the Secondary Controller Logic to automatically accept the **GETACCCR** CCC. This may be done either at the time of enabling, or at a subsequent time.

- 4365 • **Option 4: Role Detection with Dead Bus Recovery Mechanism** (i.e., will determine whether Host  
4366 Controller should start in Active Controller mode or Standby Controller mode):  
4367   A. The Driver initializes both the I3C Bus Controller Logic and the I3C Secondary Controller Logic,  
4368   but does not write to field **BUS\_ENABLE** in register **HC\_CONTROL** (*Section 7.4.2*) or either of the  
4369   above mentioned fields in register **STBY\_CR\_CONTROL** (*Section 7.7.11.1*).  
4370   B. The Driver writes an appropriate value into the registers that configure the Dead Bus Recovery  
4371   Mechanism (see *Section 6.18*) via its Extended Capability structure (see *Section 7.7.6*). This  
4372   causes the Host Controller to engage the Dead Bus Recovery mechanism and use the Dead Bus  
4373   Recovery flow (i.e., Error Type DBR) to detect whether the I3C Bus already has an Active  
4374   Controller, and choose the correct role for the Host Controller.  
4375   C. The Host Controller begins operation in either Active Controller mode or Standby Controller  
4376   mode, as determined by the Dead Bus Recovery Mechanism.

4377 **Note:**

4378   *If the Driver initially configured the Host Controller to operate according to **Option 1** (I3C Primary  
4379   Controller Only), then the Driver may choose to subsequently initialize and enable the Secondary  
4380   Controller Logic separately from the Bus Controller Logic. This shall have the same effect as  
4381   though the Driver had used **Option 2** (I3C Primary Controller with support for the Controller-to-  
4382   Controller Handoff Procedure), as it would also enable full support for passing the Controller Role  
4383   to a Secondary Controller Device (if such support were not enabled earlier).*

4384   *Before the I3C Secondary Controller Logic is enabled, field **ACR\_FSM\_OP\_SELECT** in register  
4385   **STBY\_CR\_CONTROL** (*Section 7.7.11.1*) acts as a vector indicating the desired initial mode and role.  
4386   The Host Controller shall use the value written into this field, as well as the order in which the  
4387   Driver enables the I3C Bus Controller Logic and I3C Secondary Controller Logic, to determine the  
4388   initial mode and role on the I3C Bus. Once the I3C Secondary Controller Logic is enabled, this field  
4389   acts to enable transitions from the current mode and role while the Host Controller is active. That is,  
4390   writing a different value indicates readiness to change roles, per *Section 6.17.2* and  
4391   *Section 6.17.4*.*

### 6.17.1.1 Secondary Controller Initialization

To initialize the Host Controller to operate as a Secondary Controller, either at initialization time or in the future, the Driver must configure the Secondary Controller Logic by writing values to the appropriate registers in the Standby Controller Mode Extended Capability structure (see [Section 7.7.11](#)).

In addition to the standard Host Controller initialization steps defined in [Section 6.1](#), the Driver should also perform the following steps:

1. Evaluate the Standby Controller Mode Extended Capability structure (see [Section 7.7.11](#)):

- A. Determine whether an I3C Target Transaction Interface to software is supported.

If this interface is supported, then discover its capabilities and configure the interface appropriately (i.e., by using its vendor-specified Extended Capability structure). Then, enable the interface using field **TARGET\_XACT\_ENABLE** in register **STBY\_CR\_CONTROL** ([Section 7.7.11.1](#)).

- B. Determine which methods of Dynamic Address Assignment are implemented and supported, by reading register **STBY\_CR\_CAPABILITIES** ([Section 7.7.11.3](#))

- C. Determine which other CCCs need to be supported and configured, either as registers in the Standby Controller Mode Extended Capability structure, or as registers in any instances of other vendor-specific Extended Capability structures that might be defined to provide control for such CCC handling.

2. Set the **BCR** bits and the **DCR** value in register **STBY\_CR\_DEVICE\_CHAR** ([Section 7.7.11.5](#)).

3. Optionally, set the **PID** value in registers **STBY\_CR\_DEVICE\_CHAR** and **STBY\_CR\_DEVICE\_PID\_LO** ([Section 7.7.11.5](#) and [Section 7.7.11.6](#)) if required for the Dynamic Address Assignment with **ENTDAA** procedure, or if it is known that the **GETPID** CCC will be used by an Active Controller.

4. Configure registers to set up autonomous responses for CCCs, for those CCCs that are defined for such handling in this specification (see [Section 6.17.3.1](#)).

5. Enable Secondary Controller Interrupts:

In register **STBY\_CR\_INTR\_SIGNAL\_ENABLE** ([Section 7.7.11.8](#)), set the mask of enabled interrupts.

These steps might need to be done at different times, depending on the Host Controller's role when it begins operation on the I3C Bus. Additionally, some steps might be omitted or delayed (i.e., might be executed after the steps that initialize and enable the Secondary Controller Logic), depending on the initial role and use case.

After completing the initialization of the Secondary Controller Logic, the Driver may set field **STBY\_CR\_ENABLE\_INIT** in register **STBY\_CR\_CONTROL** ([Section 7.7.11.1](#)) to enable the Secondary Controller Logic. Per [Section 6.17.1](#), the value written to this field, and the order in which the Bus Controller Logic and the Secondary Controller Logic are enabled by the Driver, shall determine the Host Controller's initial role on the I3C Bus, as well as the initial operating mode.

### 6.17.1.2 Configuration in Active Controller Mode

If the Host Controller is initialized in Active Controller mode and, and if it wishes to control the I3C Bus as the Primary Controller, then the Driver should first perform all of the steps in the Host Controller initialization procedure as defined in *Section 6.1.1*, and then perform all the steps of the Secondary Controller Initialization procedure as defined in *Section 6.17.1.1*.

Following that, the Driver must configure the Host Controller to have a Dynamic Address by writing to the appropriate fields of register **CONTROLLER\_DEVICE\_ADDR** (*Section 7.4.3*).

**Note:**

*For this configuration, The Driver should not attempt to configure a Static Address in register **STBY\_CR\_DEVICE\_ADDR** (*Section 7.7.11.2*) because the Driver assigns the initial Dynamic Address. However, after passing the Controller Role, another Controller-capable Device might change or reset this Dynamic Address, so the Driver should enable at least one supported mode for Dynamic Address Assignment in Standby Controller mode (per *Section 6.17.1.3*). For such situations, Dynamic Address Assignment with the **ENTDAA** procedure is recommended.*

The Driver should then:

1. Start the Host Controller in Active Controller mode.
2. Write a value of 1'b1 to field **ACR\_FSM\_OP\_SELECT** in register **STBY\_CR\_CONTROL** (*Section 7.7.11.1*) to set the initial mode and role as Active Controller.
3. Enable the Secondary Controller Logic by writing a value of 2'b01 to field **STBY\_CR\_ENABLE\_INIT** in register **STBY\_CR\_CONTROL** (*Section 7.7.11.1*).

**Note:**

*The Host Controller shall begin operation in Active Controller mode, since the I3C Bus Controller Logic was enabled first, and since field **ACR\_FSM\_OP\_SELECT** was set to 1'b1 when the Secondary Controller Logic was subsequently enabled. Because the Host Controller is initially operating in Active Controller mode, the Secondary Controller Logic must start in a standby state. The Secondary Controller Logic shall remain in standby (not in the transitional state per *Section 6.17.2*) until the Driver enters the transitional state and subsequently issues the special Command Descriptor to send the **GETACCCR** CCC, initiating the Controller-to-Controller Handoff Procedure. If those steps are successful, then the Controller Role will have been successfully passed to another Controller-capable Device on the I3C Bus; only then would the Secondary Controller Logic be active.*

### 6.17.1.3 Initialization and Dynamic Address Assignment in Standby Controller Mode

If the Host Controller is initialized in Standby Controller Mode and wishes to join the I3C Bus as a Secondary Controller, then the Driver must configure it to participate in Dynamic Address Assignment so that the I3C Bus' Active Controller can assign a Dynamic Address (see the I3C Specification [*MIPI02*] at *Section 5.1.4.2*).

For the initialization and process of joining the I3C Bus in Standby Controller mode, the Driver must perform the following steps:

1. Perform steps 1–10 in the Host Controller Initialization procedure (per *Section 6.1.1*), but stop before enabling the Host Controller via register **HC\_CONTROL** (i.e., stop before step 11). Refer to *Section 6.17.1* for details on how the order in which the I3C Bus Controller Logic and I3C Secondary Controller Logic are enabled determines the Host Controller's initial mode and role.
2. Perform all of the steps in the Secondary Controller Initialization procedure per *Section 6.17.1.1*.
3. Enable at least one supported method for Dynamic Address Assignment, using register **STBY\_CR\_CONTROL** (*Section 7.7.11.1*).

If using any of the methods that rely on Static Addresses (i.e., **SETDASA** or **SETAASA**), then also program the assigned Static Address into field **STATIC\_ADDR** in register

**STBY\_CR\_DEVICE\_ADDR** (*Section 7.7.11.1*), and also set its field **STATIC\_ADDR\_VALID** to 1'b1.

Note that knowledge of the Static Address must also be programmed into the Primary Controller.

4. Write a value of 1'b0 to field **ACR\_FSM\_OP\_SELECT** in register **STBY\_CR\_CONTROL** (*Section 7.7.11.1*) to set the initial mode and role as Secondary Controller.
5. Enable the Secondary Controller Logic by writing to field **STBY\_CR\_ENABLE\_INIT** in register **STBY\_CR\_CONTROL** (*Section 7.7.11.1*).
  - A. If the Secondary Controller Logic supports sending a Hot-Join Request, then it shall also support Dynamic Address Assignment with the **ENTDAA** CCC. That is, fields **DAA\_ENTDAA\_SUPPORT** in register **STBY\_CR\_CAPABILITIES** (*Section 7.7.11.3*) and **DAA\_ENTDAA\_ENABLE** in register **STBY\_CR\_CONTROL** (*Section 7.7.11.1*) shall both have a value of 1'b1.

To use Hot-Join, the Driver shall write a value of 2'b11 to field **STBY\_CR\_ENABLE\_INIT** in register **STBY\_CR\_CONTROL** (*Section 7.7.11.1*). This shall enable the Secondary Controller Logic, and trigger it to emit the Hot-Join Request, once it has determined that it has met the conditions of eligibility as a Hot-Joining Target, as defined by the I3C Specification at *Section 5.1.5* [*MIPI02*]. Note that the Driver must not assign a Dynamic Address to the Host Controller (i.e., field **DYNAMIC\_ADDR\_VALID** in register **STBY\_CR\_DEVICE\_ADDR** must have a value of 1'b0).

While the Secondary Controller Logic is still waiting to emit its first Hot-Join Request, field **STBY\_CR\_ENABLE\_INIT** in register **STBY\_CR\_CONTROL** (*Section 7.7.11.1*) shall return a value of 2'b11 on a subsequent read.

Once the Secondary Controller Logic has successfully emitted its first Hot-Join Request (which the Active Controller might respond to with either an ACK or a NACK), field **STBY\_CR\_ENABLE\_INIT** shall return a value of 2'b10 on a subsequent read. Additionally, field **HJ\_REQ\_STATUS** in register **STBY\_CR\_STATUS** (see *Section 7.7.11.4*) shall return a value of 1 on a subsequent read.

**Note:**

*Since Dynamic Address Assignment may happen at some time after the Hot-Join Request, the Driver should watch for a subsequent interrupt that signals a successful assignment of a Dynamic Address, with interrupt **STBY\_CR\_DYN\_ADDR\_STAT** (see *Section 7.7.11.7*).*

- B. If Hot-Join Requests are not supported, or if the Driver does not wish to use Hot-Join, then the Driver shall write 2'b10 to field **STBY\_CR\_ENABLE\_INIT** in register **STBY\_CR\_CONTROL** (*Section 7.7.11.1*). This shall enable the Secondary Controller Logic, which will then wait

4505 passively for Dynamic Address Assignment, using any of the methods that are both supported in  
4506 register **STBY\_CR\_CAPABILITIES** and enabled in register **STBY\_CR\_CONTROL**. Assigning a  
4507 Dynamic Address from a Static Address requires the Driver to write a valid Static Address to field  
4508 **STATIC\_ADDR** in register **STBY\_CR\_DEVICE\_ADDR**, and also requires the Active Controller (i.e., the  
4509 Primary Controller) to know this Address via other means.

4510 **Note:**

4511     *This I3C HCI Specification does not define a method to inform the Primary Controller of this*  
4512     *Static Address.*

- 4513     6. Subsequently, enable the Bus Controller Logic per steps 11 and higher (as applicable) of the Host  
4514       Controller Initialization procedure defined in *Section 6.1.1*.

4515     **Note:**

4516         *At the time that the Secondary Controller Logic was enabled, field BUS\_ENABLE in register*  
4517         *HC\_CONTROL was not yet set to 1'b1 (per step 1 above) and field ACR\_FSM\_OP\_SELECT was*  
4518         *set to 1'b0. As a result, the Host Controller shall begin operation in Standby Controller mode,*  
4519         *with a Secondary Controller role on the I3C Bus.*

4520         *If the Driver does not subsequently enable the Bus Controller Logic, then the Host Controller*  
4521         *will not be able to accept the Controller Role from the Active Controller, and will be unable to*  
4522         *transition into Active Controller mode. In such a state, the Host Controller shall not allow the*  
4523         *Secondary Controller Logic to be primed to accept the GETACCCR CCC.*

4524     After the Driver writes to field **STBY\_CR\_ENABLE\_INIT**, it shall wait for a subsequent  
4525       **STBY\_CR\_DYN\_ADDR\_STAT** interrupt (read from register **STBY\_CR\_INTR\_STATUS** [*Section 7.7.11.3*]) for  
4526       notification that a Dynamic Address has been successfully assigned.

4527     At any time after initial Dynamic Address Assignment, the Active Controller may reconfigure the Dynamic  
4528       Address using the **SETNEWDA** CCC, or reset the Dynamic Address using the **RSTDAA** CCC. Changes to  
4529       the Dynamic Address will be reflected in field **DYNAMIC\_ADDR** in register **STBY\_CR\_DEVICE\_ADDR**  
4530       (*Section 7.7.11.2*). If the Active Controller uses the **RSTDAA** CCC to reset the Dynamic Address, then the  
4531       **DYNAMIC\_ADDR\_VALID** field will read as 1'b0 until the Active Controller sets a new Dynamic Address using  
4532       a supported method. Any such changes of the Dynamic Address or its validity may generate subsequent  
4533       **STBY\_CR\_DYN\_ADDR\_STAT** interrupts (read from register **STBY\_CR\_INTR\_STATUS**) to notify the Host that  
4534       configuration changes have been made.

4535     This shall also apply to a Host Controller that was formerly an Active Controller (e.g., the Primary  
4536       Controller, as shown in *Section 6.17.1.2*) and subsequently passed the Controller Role to another  
4537       Controller-capable Device on the I3C Bus.

4538     **Note:**

4539         *Many of the fields for configuration registers in the Standby Controller Mode Extended Capability*  
4540         *structure (*Section 7.7.11*) cannot be changed while the Host Controller is enabled in Standby*  
4541         *Controller mode (i.e., if field STBY\_CR\_ENABLE\_INIT in register STBY\_CR\_CONTROL is set to a non-*  
4542         *zero value). In such a state, the Host Controller shall block writes to these fields. These fields may*  
4543         *only be changed while the Host Controller is disabled, since such changes would not be known to*  
4544         *the other Active Controller, and such a change might violate a private contract or create a mismatch*  
4545         *in expectations.*

### 6.17.2 Transition from Active Controller Mode to Standby Controller Mode

To transition the Host Controller into Standby Controller mode, the Driver shall perform the following high-level procedures:

1. Determine the appropriate steps to prepare the I3C Bus for the Controller-to-Controller Handoff Procedure, per [Section 5.1.7.1](#) of the I3C Specification [[MIPI02](#)].
2. Prepare the Host Controller for handoff (see [Section 6.17.2.1](#)).
3. Once the Host Controller is ready to transition, enqueue the special Command Descriptor to transition to Standby Controller mode (see [Section 6.17.2.2](#)).

This special Command Descriptor is a sub-command of the Internal Control Command, which may only be used in this transitional state. The sub-command sends the **GETACCCR** CCC and automatically initiates the Controller-to-Controller Handoff Procedure, using the I3C Bus Controller Logic and I3C Secondary Controller Logic within the Host Controller (see [Section 8.4.2.7](#)).

4. Remain ready to receive the Controller Role again, if any of the following events occur:
  - A. If the Secondary Controller Logic detects an Error Type CE3 failure and initiates an automatic return to Active Controller mode, due to the chosen Secondary Controller (i.e., the new Active Controller) failing to assert its Controller Role status, after a handoff that was initially presumed to be successful. See the I3C Specification [[MIPI02](#)] at [Section 5.1.10.2.4](#).
  - B. If the new Active Controller subsequently sends the **GETACCCR** CCC. (If this Host Controller's I3C Secondary Controller Logic is primed to automatically accept the CCC, then per [Section 6.17.4](#) it shall do so.)

In the Active Controller mode, the I3C Bus Controller Logic may be configured to accept a Controller Role Request (CRR) when the **CRR\_REJECT** field in the relevant DAT entry for the Target is cleared. Once the Controller Role Request is accepted from a specific Secondary Controller Device, the Driver is then responsible for initiating the above high-level procedures for transitioning to Standby Controller mode. The Host Controller shall not automatically initiate the steps listed above to send the **GETACCCR** CCC in response to a Controller Role Request. The Driver may also use the above procedures to send the **GETACCCR** CCC using the special Command Descriptor without previously receiving any Controller Role Request from a Secondary Controller Device.

Before passing the Controller Role to the chosen Secondary Controller Device, the Driver may announce the Bus configuration with the **DEFTGTS** Broadcast CCC (and other **DEFGRPA** Broadcast CCCs if applicable). If this configuration has not been announced since any changes to such Bus configuration, then this announcement before passing the Controller Role is mandatory, and the Driver must then verify that the chosen Secondary Controller Device has received and processed the data in the announced configuration data, and is ready to accept the Controller Role.

**Note:**

When sending the **DEFTGTS** Broadcast CCC, software is responsible for assembling the CCC's data message in the correct format (see the I3C Specification [[MIPI02](#)] at [Section 5.1.9.3.7](#)) and enqueueing the CCC as a Command Descriptor. The data message must include a descriptor for the Active Controller, including its Dynamic Address, DCR, BCR, and Static Address (i.e., 7'h7E) as defined for the **DEFTGTS** Broadcast CCC. The Host Controller and its Bus Controller Logic will not generate this Broadcast CCC automatically.

### 6.17.2.1 Preparing the Host Controller for Controller Role Handoff

Before entering the special transitional state, the Driver shall either clear Rings/Queues from all commands, or else work to process any enqueued Transfer Commands while blocking new Transfer Commands. The Driver shall also disable In-Band Interrupt Requests, including Controller Role Requests and Hot-Join Requests from all Target Devices other than the chosen Secondary Controller Device. The Driver shall also take any other steps necessary to prepare the I3C Bus for passing the Controller Role, per *Section 5.1.7.1* of the I3C Specification [**MIPI02**].

The transition to Standby Controller mode is explicitly enabled by writing the value 1'b0 to field **ACR\_FSM\_OP\_SELECT** in register **STBY\_CR\_CONTROL** (see *Section 7.7.11.1*), however this is usually only the last step the Driver takes before sending the special Command Descriptor to transition to Standby Controller mode. Several other key requirements must be met before the Driver writes this value to prepare for the transition.

The Host Controller shall ensure that all of the following necessary pre-Handoff conditions have been met:

1. Secondary Controller Logic has been configured per *Section 6.17.1.2*.

This includes setting field **STBY\_CR\_ENABLE\_INIT** in register **STBY\_CR\_CONTROL** to a non-zero value (*Section 7.7.11.1*)

2. The Host Controller has been configured with a valid Dynamic Address that has been broadcast to the chosen Secondary Controller Device, using the **DEFTGTS** CCC.
  - A. In register **CONTROLLER\_DEVICE\_ADDR** (*Section 7.4.3*), field **DYNAMIC\_ADDR\_VALID** must be set to 1'b1 and field **DYNAMIC\_ADDR** must be set to a Dynamic Address that is valid for an I3C Device. **This Dynamic Address will become the Host Controller's Dynamic Address** once it successfully passes the Controller Role.

**Note:**

*Software should have previously sent the DEFTGTS CCC to announce the list of all Devices (including the Dynamic Address of this Host Controller as the Active Controller) to any Secondary Controller Devices on the I3C Bus, to ensure that the chosen Secondary Controller Device knows the current configuration (see I3C Specification [**MIPI02**] at *Section 5.1.9.3.7*).*

3. The active Command Queue/Ring must be otherwise empty, and any other enabled Command Queues/Rings shall either be fully drained, quiesced or otherwise disabled, to prevent any interruptions.
  - A. **For DMA Mode:** If the Host Controller supports multiple Ring Bundles (per *Section 6.6.5*), then the Command/Response Ring Pairs for such Ring Bundles that are enabled and running should be disabled or halted; or such Ring Pairs for all Ring Bundles except one (i.e., except for the Ring Pair that would be used in following steps) should be disabled, such that only one Ring Bundle's Command/Response Ring Pair is both **ENABLED** and **RUNNING**. The Host Controller should also consume any enqueued Response Descriptors from such enabled Response Rings.
  - B. **For PIO Mode:** The Host Controller should reset or empty the Command Queue, or allow it to drain to an empty state. The Host Controller should also consume any outstanding Response Descriptors along with any associated Rx Data from the Rx Data Buffer (per *Section 6.8.1*) until all such PIO Queues are empty.
  - C. If the Host Controller supports Scheduled Commands (per *Section 6.16*), then this capability should be disabled to prevent any interruptions.
  - D. If the Host Controller supports any other implementer-defined capability that could cause interrupts to the processing of I3C commands, if it were to be enabled, then such capability should also be disabled.

Once all of these conditions are true, and as long as they remain true, the Driver may write a value of 1'b0 to field **ACR\_FSM\_OP\_SELECT**. This signals readiness to transition into Standby Controller mode and allows the Driver to subsequently send the special Command Descriptor (per *Section 6.17.2.2*) that sends the **GETACCR** CCC and initiates the Controller-to-Controller Handoff Procedure.

4636

**Note:**

4637

If the Host Controller accepts the value of 1'b0 written to field **ACR\_FSM\_OP\_SELECT** in register **STBY\_CR\_CONTROL**, then it has entered the transitional state and must not accept any Command Descriptor other than the special Internal Control Command sub-type to send the **GETACCCR CCC** (see **Section 6.17.2.2**).

4641

If the necessary pre-handoff conditions have not been met, then the Host Controller may reject the value of 1'b0 written to field **ACR\_FSM\_OP\_SELECT**. The Driver should verify that this value has been accepted, by reading register **STBY\_CR\_CONTROL**. If field **ACR\_FSM\_OP\_SELECT** returns a value of 1'b1 after writing a value of 1'b0, then the Host Controller was not ready to begin the transition and is not in the transitional state, and the Driver should check the pre-handoff conditions.

### 6.17.2.2 Sending the GETACCCR CCC and Initiating the Controller Role Handoff Procedure

In Active Controller mode, a value of 1'b0 read from field **ACR\_FSM\_OP\_SELECT** indicates that the Host Controller is ready to accept the special Command Descriptor on the Queue/Ring, which would send the **GETACCCR** CCC to the chosen Secondary Controller Device.

To send the **GETACCCR** CCC and initiate the Controller Role Handoff Procedure, the Driver shall construct the Internal Control Command using the appropriate sub-command type as defined in *Section 8.4.2.7*, and then enqueue the Command Descriptor to the active Command Queue/Ring for processing.

As noted in *Section 6.17.2.1*, the Host Controller shall only accept this special Command Descriptor while in this transitional state, and it shall not accept any other types of Command Descriptors (i.e., any other Transfer Commands).

If the Driver enqueues this special Command Descriptor while the Host Controller is **not** in this transitional state, then the Host Controller shall reject the Command Descriptor as an error, and shall generate a Response Descriptor with the value 0xA (**NOT\_SUPPORTED**) in field **ERR\_STATUS** (per *Table 143*).

If the Host Controller receives any other type of Command Descriptor while in this transitional state, then it shall reject the Command Descriptor as an error, and shall generate a Response Descriptor with the value 0x8 (**HC\_TERMINATED**) in field **ERR\_STATUS** (per *Table 143*). The Host Controller shall also leave the transitional state, and field **ACR\_FSM\_OP\_SELECT** shall have a value of 1'b1.

The Driver shall prepare such a Command Descriptor, and either:

- **For PIO Mode:** Write the Command Descriptor into the Command Queue Port; or
- **For DMA Mode:** Construct a Transfer Descriptor for the Command Descriptor, and enqueue it into the Ring Bundle that will be used for the transition. (In most cases, this will typically be the sole remaining Ring Bundle that is both **ENABLED** and **RUNNING**.)

Once the Host Controller receives this special Command Descriptor, it shall ensure that the necessary conditions were met, and that the Host Controller is in the transitional state. If the conditions are met and the transitional state is still in effect, then the Host Controller shall latch the current value of field **PRIME\_ACCEPT\_GETACCCR** in register **STBY\_CR\_CONTROL** (*Section 7.7.11.1*). The latched value shall be used to determine whether the I3C Secondary Controller Logic will automatically be primed to accept the Controller Role after successfully transitioning to Standby Controller mode.

- If field **PRIME\_ACCEPT\_GETACCCR** was set to 1'b0 when the Host Controller accepted the special Command Descriptor, then:
  - The Host Controller shall not prime the Secondary Controller Logic to automatically accept the Controller Role from the Active Controller after a successful transition to Standby Controller mode.
  - The Host Controller shall notify the Host upon successful transition, with an interrupt of type **ACR\_HANDOFF\_OK\_REMAIN\_STAT**.
  - However, the Driver may subsequently write a value of 1'b1 to field **ACR\_FSM\_OP\_SELECT** (per *Section 6.17.4*) after receiving notification of a successful transition.
- If field **PRIME\_ACCEPT\_GETACCCR** was set to 1'b1 when the Host Controller accepted the special Command Descriptor, then:
  - The Host Controller shall prime the Secondary Controller Logic to automatically accept the Controller Role from the Active Controller (i.e., if the Active Controller sends the **GETACCCR** CCC to this Host Controller's Dynamic Address).
  - This shall take effect only upon a successful transition into Standby Controller mode that is not otherwise cancelled (i.e., if the Secondary Controller Logic initiates an automatic return to Active Controller mode, due to an Error Type CE3 failure).
  - The Host Controller shall notify the Host upon successful transition, with an interrupt of type **ACR\_HANDOFF\_OK\_PRIMED\_STAT** to indicate that the Secondary Controller Logic is primed.

- 4693     • After this interrupt notification, field **ACR\_FSM\_OP\_SELECT** shall have a value of 1'b1 (i.e., as though  
4694       the Driver had written a value of 1'b1 to field **ACR\_FSM\_OP\_SELECT** upon notification of successful  
4695       transition). The Driver may subsequently revoke readiness (i.e., by writing a value of 1'b0 per  
4696       **Section 6.17.4**) if this field is writeable while the Host Controller is operating in Standby Controller  
4697       mode.

4698     After the Host Controller latches the primed state and finishes all preliminary checks, it shall prepare the  
4699       Secondary Controller Logic for activation, and also drive the Bus Controller Logic to send the **GETACCCR**  
4700       CCC to the chosen Secondary Controller Device using the Dynamic Address specified in the special  
4701       Command Descriptor.

4702     **Note:**

4703       *If the Host Controller rejects the Command Descriptor due to incorrect format, then field  
4704       **ACR\_FSM\_OP\_SELECT** shall revert to the value 1'b1. Before sending a subsequent Command  
4705       Descriptor for the **GETACCCR** CCC, the software must first set this bit to 1'b0 again. However, field  
4706       **ACR\_FSM\_OP\_SELECT** shall not revert to the value 1'b1 as a result of the Host Controller rejecting  
4707       a Command Descriptor for other reasons (i.e., necessary pre-Handoff conditions not being true).*

4708     Because the Bus Controller Logic drives the **GETACCCR** CCC and manages the Controller-to-Controller  
4709       Handoff Procedure, it shall monitor the I3C Bus to determine whether the Handoff is successful. The status  
4710       of the Handoff Procedure shall be returned in the Response Descriptor.

- 4711       • If the chosen Secondary Controller does not ACK its Dynamic Address (i.e., passive NACK) during the  
4712        Direct CCC framing, then the **GETACCCR** CCC will fail per the I3C Specification [**MIPI02**] at  
4713        **Section 5.1.9.3.16, Figure 63 (GETACCCR Format 2: Not Accepted)**.
  - 4714           • If this occurs, then the Bus Controller Logic shall cancel the CCC by sending a Repeated START,  
4715            followed by the I3C Broadcast Address, etc. The Response Descriptor would indicate this result with  
4716            the value 0x5 (**NACK**) in field **ERR\_STATUS**.
  - 4717           • If the chosen Secondary Controller does ACK its Dynamic Address during the Direct CCC framing, but  
4718            the Bus Controller Logic detects an incorrect response on the SDA Line for the returned Dynamic  
4719            Address and Parity Bit, then the **GETACCCR** CCC must fail per the I3C Specification [**MIPI02**] at  
4720            **Section 5.1.9.3.16, Figure 64 (GETACCCR Format 3: Incorrect Cancel)**.
    - 4721              • If this occurs, then the Bus Controller Logic shall cancel the CCC by sending a Repeated START,  
4722              followed by the I3C Broadcast Address, etc. The specific status in the Response Descriptor shall  
4723              depend on the mode of failure, based on the response on the SDA Line:
      - 4724               • If the I3C Bus Controller Logic reads the correct Dynamic Address with an incorrect Parity Bit, then  
4725                the Response Descriptor shall indicate this result with the value 0xC (**Transfer Type Specific:  
4726                GETACCCR\_PARITY**) in field **ERR\_STATUS**.
      - 4727               • If the I3C Bus Controller Logic reads an incorrect Dynamic Address (which might be caused by  
4728                interference on the SDA Lane), then the Response Descriptor shall indicate this result with the value  
4729                0xD (**Transfer Type Specific: GETACCCR\_DYNADDR**) in field **ERR\_STATUS**.
      - 4730               • These error codes are defined in **Section 6.13.1** and **Table 143**.
    - 4731              • If the chosen Secondary Controller does ACK its Dynamic Address during the Direct CCC framing, and  
4732              if the Bus Controller Logic detects a correct response on the SDA Line for the returned Dynamic Address  
4733              and Parity Bit, then the **GETACCCR** CCC will be successful per the I3C Specification [**MIPI02**] at  
4734              **Section 5.1.9.3.16, Figure 62 (GETACCCR Format 1: Accepted)**.

4735     In this case, the I3C Bus Controller Logic shall drive a STOP condition at the end of the CCC, and  
4736       follow with the Controller-to-Controller Handoff Procedure per the I3C Specification [**MIPI02**] at  
4737       **Section 5.1.7.2**, as well as a test for Error Type CE3. The Response Descriptor shall indicate this  
4738       result with the value 0x0 (**SUCCESS**) in field **ERR\_STATUS**.

4739     If the transaction is successful, then software shall read the Response Descriptor as the last action in Active  
4740       Controller mode. If the transaction is not successful, then field **ACR\_FSM\_OP\_SELECT** shall keep its value  
4741       of 1'b0, and software may either leave it at this value (i.e., remaining in the transitional state, and possibly  
4742       enqueueing another special Command Descriptor to try sending **GETACCCR** CCC again), or else write a

4743 value of 1'b1 to send Command Descriptors for any other purpose, such as Transfer Commands. For  
4744 unsuccessful transactions, the Host Controller shall also trigger an interrupt to notify the Driver, per **Table**  
4745 **9**.

4746 Once the **GETACCCR** CCC is sent successfully and the Controller Role has been passed (which should  
4747 occur on the subsequent STOP condition on the I3C Bus), the Bus Controller Logic shall stop driving the  
4748 SCL clock, and the Secondary Controller Logic shall await either incoming requests or assertions of the  
4749 Controller Role from the new Active Controller.

4750 After attempting to pass the Controller Role, the Host Controller shall report successful transitions or failed  
4751 transition attempts from Active Controller mode to Standby Controller mode by triggering one or more of  
4752 the interrupt events listed in **Table 9** in register **STBY\_CR\_INTR\_STATUS** (**Section 7.7.11.7**).

4753 **Table 9 Defined Interrupt Events Reported for Controller Role Handoff Procedure**

Interrupt Event	Description
<b>ACR_HANDOFF_OK_REMAIN_STAT</b>	The Controller Role Handoff was initially successful. If it remains successful (i.e., if no Error Type CE3 condition is subsequently detected), then the Host Controller has transitioned to Standby Controller mode. The I3C Secondary Controller Logic shall not automatically accept the Controller Role.
<b>ACR_HANDOFF_OK_PRIMED_STAT</b>	The Controller Role Handoff was initially successful. If it remains successful (i.e., if no Error Type CE3 condition is subsequently detected), then the Host Controller has transitioned to Standby Controller mode. The I3C Secondary Controller Logic shall automatically accept the <b>GETACCCR</b> CCC sent by the new Active Controller (per <b>Section 6.17.4</b> ), unless conditions for readiness are no longer true.
<b>ACR_HANDOFF_ERR_FAIL_STAT</b>	The Controller Role Handoff was not successful due to an error. The Host Controller remains in Active Controller mode. Response Descriptor field <b>ERR_STATUS</b> shall indicate more detailed failure status (see <b>Table 143</b> ).
<b>ACR_HANDOFF_ERR_M3_STAT</b>	The Controller Role Handoff was initially successful, but the I3C Secondary Controller Logic subsequently detected an Error Type CE3 condition. As a result, the Host Controller has returned to Active Controller mode. The I3C Bus Controller Logic is now active.

4754

**Note:**

4755

The interrupt events in **Table 9** describe attempted transitions from Active Controller mode into Standby Controller mode. By contrast, the interrupt events in **Table 10** describe attempted transitions in the other direction (i.e., from Standby Controller mode into Active Controller mode).

4756

In cases where a single Controller Role Handoff Procedure is quickly followed by another transition back into Active Controller Mode (i.e., if the Secondary Controller Logic was primed to accept the Controller Role and the new Active Controller subsequently passed the Controller Role back after it completed its tasks), software should expect to see at least one interrupt event from both **Table 9** and **Table 10**: one successful transition event for each direction. Depending on system interrupt latency and the timing of the two Controller Role Handoff Procedures on the I3C Bus, both such interrupt events might arrive within a very short time interval, and it's possible that the software might read these events with a single read of register **STBY\_CR\_INTR\_STATUS**, with no detectable change in the value of field **AC\_CURRENT\_OWN** between the two interrupt events.

4757

If interrupts **ACR\_HANDOFF\_OK\_REMAIN\_STAT** and **ACR\_HANDOFF\_ERR\_M3\_STAT** were both triggered, then that indicates that the Host Controller attempted to pass the Controller Role and remain in Standby Controller mode (i.e., with Secondary Controller Logic not in primed state per **Section 6.17.4**), but the Secondary Controller Logic subsequently detected an Error Type CE3 condition after Handoff and was forced to return the Host Controller to Active Controller mode.

4758

If interrupts **ACR\_HANDOFF\_OK\_PRIMED\_STAT** and **ACR\_HANDOFF\_ERR\_M3\_STAT** were both triggered, then that indicates that the Host Controller attempted to pass the Controller Role and remain ready to accept the Controller Role (i.e., with Secondary Controller Logic in primed state per **Section 6.17.4**), but the Secondary Controller Logic subsequently detected an Error Type CE3 condition after Handoff and was forced to return the Host Controller to Active Controller mode.

4759

Both of these situations should typically be considered errors (or exceptions to the standard flow), because the intended new Active Controller did not assert its Controller Role status, or failed to respond to a START Request in a timely manner. Note that in such situations the Response Descriptor for the special Command Descriptor will return a value of 0x0 in field **ERR\_STATUS** because it only reports the initially successful handoff, not the subsequent Error Type CE3 condition.

4760

The Driver should also watch for the transition status bus monitoring bit, **AC\_CURRENT\_OWN**, to change to 1'b0. But an interrupt notification is the definitive method of determining whether a transition was successful or not successful, and (in some cases) whether it was also accompanied by a subsequent transition back into Active Controller mode.

4761

The transition shall trigger an interrupt notification to report the status:

4762

- An interrupt notification of either type **ACR\_HANDOFF\_OK\_PRIMED\_STAT** or type **ACR\_HANDOFF\_OK\_REMAIN\_STAT** signals that the Host Controller is now operating in Standby Controller mode, or was initially and transiently in that state:
  - If an interrupt notification of type **ACR\_HANDOFF\_ERR\_M3\_STAT** is also triggered, then this signals that the Host Controller was forced to return to Active Controller mode (see below).
  - However, if no interrupt notification of type **ACR\_HANDOFF\_ERR\_M3\_STAT** is triggered, and if the new Active Controller has asserted its Controller Role, then the Host Controller shall remain in Standby Controller mode per **Section 6.17.3** (i.e., no Error Type CE3 condition is detected); and may also conditionally remain ready to accept the Controller Role per **Section 6.17.4**. The new Active Controller may subsequently attempt to pass back the Controller Role, at its discretion.
- An interrupt notification of type **ACR\_HANDOFF\_ERR\_FAIL\_STAT** signals that the Controller Role Handoff Procedure was not successful (as noted in **Table 9**) and the Host Controller is still operating in Active Controller mode.

4763

In this case, the Host Controller would still be in the transitional state, and the **AC\_CURRENT\_OWN** bit would read as 1'b0. The Driver should read this status, attempt to determine why the Controller Role Handoff Procedure was not successful, and optionally initiate another attempt to send the **GETACCCR CCC**.

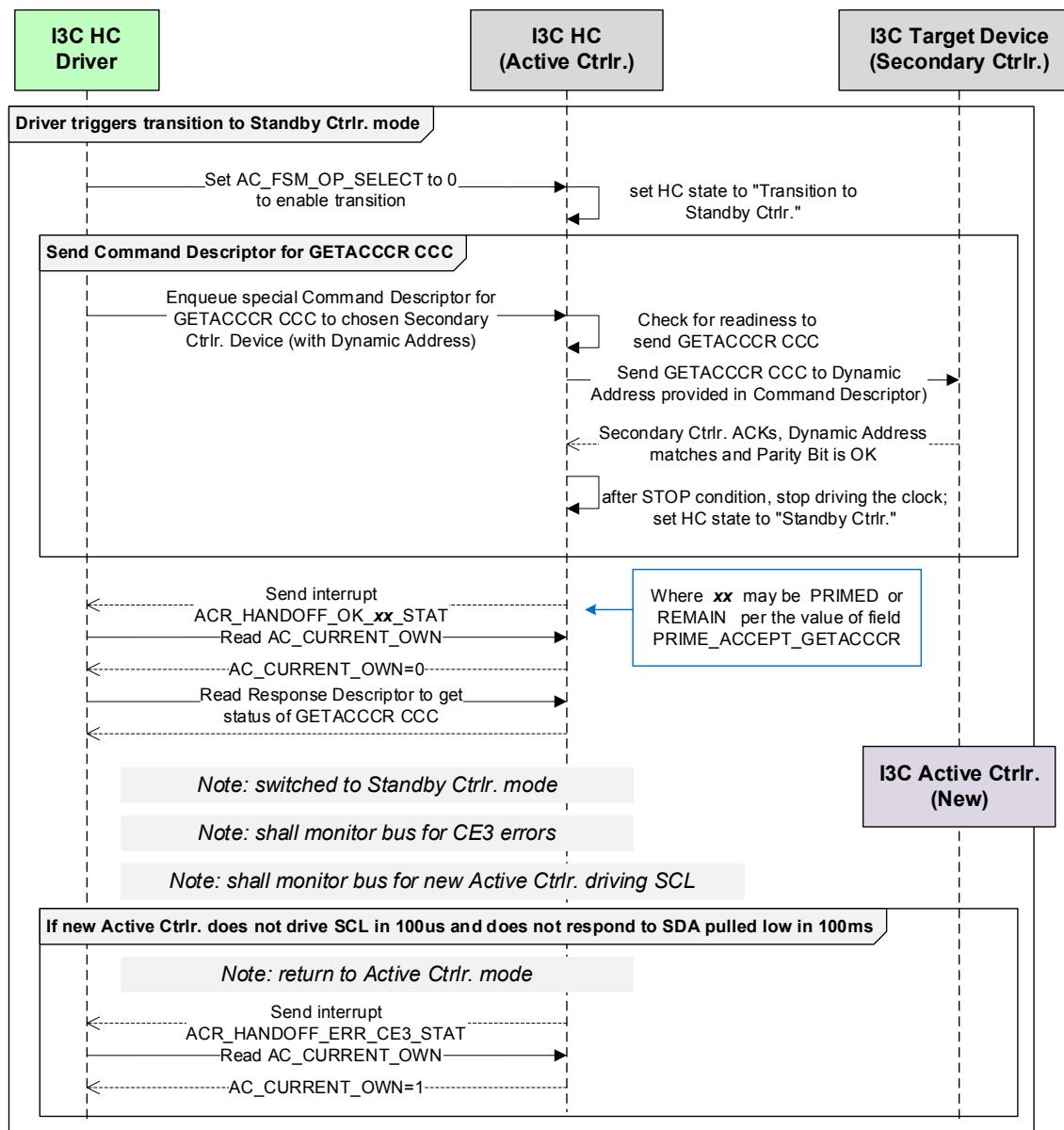
- 4805 • An interrupt notification of type **ACR\_HANDOFF\_ERR\_M3\_STAT** signals an error: the Secondary  
4806 Controller Logic detected an Error Type CE3 condition per **Section 5.1.10.2.4** of the I3C Specification  
4807 [**MIPI02**].

4808 In this case, the Host Controller would have automatically returned to Active Controller mode and  
4809 the **AC\_CURRENT\_OWN** bitfield in register **STBY\_CUR\_STATUS** (**Section 7.7.11.4**) would read as 1'b1.  
4810 The Driver should read this status, then run an error recovery procedure to determine why the  
4811 Controller Role Handoff Procedure was not successful or why the new Active Controller was unable  
4812 to assert its Controller Role status. The Driver may subsequently initiate another attempt to send the  
4813 **GETACCCR** CCC.

4814 **Note:**

4815 *To run an error recovery procedure, or any other CCC (except **GETACCCR**), or to cancel the  
4816 transition to Standby Controller mode, software must first write the value 1'b1 to field  
4817 **ACR\_FSM\_OP\_SELECT** in register **STBY\_CR\_CONTROL** (**Section 7.7.11.1**). (Software may confirm  
4818 this by subsequently reading the register to verify that the value 1'b1 is read.) After writing this 1'b1,  
4819 software may send regular Command Descriptors, but software shall not send the special  
4820 Command Descriptor for the **GETACCCR** CCC.*

4821 **Figure 36** shows the flow for enabling the transition, sending the special Command Descriptor for the  
4822 **GETACCCR** CCC, and successfully completing the Controller-to-Controller Handoff Procedure. This flow  
4823 assumes that all needed steps to prepare for Handoff have been completed in advance.



4824

**Figure 36 Flow to Transition to Standby Controller Mode**

### 6.17.3 Operation in Standby Controller Mode

In Standby Controller mode, the Host Controller supports a limited set of features and capabilities that are appropriate for the Secondary Controller role on the I3C Bus.

The following features and capabilities are required, and shall be supported if Standby Controller mode is supported:

- Secondary Controller Logic may receive the Controller Role from the Active Controller, subject to readiness conditions defined in **Section 6.17.4**.
- Secondary Controller Logic may autonomously service certain standard CCCs sent by the Active Controller, per **Section 6.17.3.1**.

**Note:**

*A Host Controller that only implements the Minimal Standby Controller definition in Section 6.17 is required to support the features and capabilities specified above. Such minimum requirements allow the Host Controller to receive the Controller Role from the Active Controller, as well as to comply with the essential requirements of the I3C Specification.*

The following features and capabilities are optional, and may be supported at the implementer's discretion:

- Support for generic Read-Type and Write-Type transactions sent by the Active Controller via an I3C Target Transaction Interface available to software, including support for CCCs that are not supported natively within the Secondary Controller Logic.
- Support for sending Controller Role Requests, for the purpose of requesting or attempting to regain the Controller Role from the Active Controller, per **Section 6.17.3.3**.

If implemented, then all such features and capabilities are part of the I3C Secondary Controller Logic, and shall be enabled in Standby Controller mode (i.e., if field **STBY\_CR\_ENABLE\_INIT** in register **STBY\_CR\_CONTROL** has a non-zero value, see **Section 7.7.8.1**), if the Host Controller was initialized per the initialization steps defined in sub-sections of **Section 6.17.1**, specifically:

- If initialized as a Secondary Controller on the I3C Bus, or
- If initialized as an Active Controller (i.e., the Primary Controller of the I3C Bus) that subsequently passed the Controller Role to another Secondary Controller Device and became a Secondary Controller on the I3C Bus.

A Host Controller may also support generic Read-Type and Write-Type transactions, via an I3C Target Transaction Interface to software as a separate capability that is linked with Standby Controller mode. If this Host Controller supports an I3C Target Transaction Interface for the I3C Secondary Controller Logic, then field **TARGET\_XACT\_SUPPORT** in register **STBY\_CR\_CAPABILITIES** (**Section 7.7.11.3**) shall have a value of 1'b1, and this interface shall be available for use by software. The Driver may choose to either enable this interface, or not enable it. To enable the I3C Target Transaction Interface, the Driver shall write a value of 1'b1 to field **TARGET\_XACT\_ENABLE** in register **STBY\_CR\_CONTROL** (**Section 7.7.8.1**).

Since the I3C Target Transaction Interface is not defined in this version of the I3C HCI Specification, it may be implemented as vendor-defined functionality, exposed via vendor-specific Extended Capabilities structures (per **Section 7.3.2**).

**Note:**

*If an I3C Target Transaction Interface is supported, then it may be enabled or disabled only while field **STBY\_CR\_ENABLE\_INIT** in register **STBY\_CR\_CONTROL** has a value of 2'b00.*

*Autonomous servicing of standard CCCs that are required for Secondary Controller operation (per **Section 6.17.3.1**) shall not depend on whether the I3C Target Transaction Interface is enabled or disabled; these capabilities are independently configured.*

### 6.17.3.1 Servicing CCCs Autonomously

In Standby Controller mode, the Host Controller's Secondary Controller Logic shall autonomously service certain Broadcast CCCs that are addressed to the I3C Bus, and certain Direct CCCs that are targeted to the Dynamic Address assigned to the Secondary Controller Logic.

The Secondary Controller Logic shall autonomously service CCC (i.e., shall receive data and shall provide auto-response), and the specific support for these CCCs may be exposed or configured via certain registers in the Extended Capability structure for Standby Controller mode (*Section 7.7.11*).

The implementer may choose which CCCs are supported, based on the use case; however, the minimum set of required CCCs specified in *Section 6.17.3.1.1* must always be supported. A Host Controller implementation may also support additional standard CCCs (per *Section 6.17.3.1.1*), as well as other standard CCCs that might be supported either by extensions to the Secondary Controller Logic (per *Section 6.17.3.1.2*), or by interaction with the software via the I3C Target Transaction Interface, if that is also supported and enabled (per *Section 6.17.3.1.3*). This allows advanced configurability for vendor-defined CCCs, at the discretion of the implementer and with assistance from software that might be developed to support such vendor-defined CCCs, according to a custom content protocol.

#### For Direct CCCs

The Secondary Controller Logic shall track the CCC modality for the I3C Mode, and shall respond only to CCCs that it does support internally (i.e., either standard or via extensions), or that it has been programmed to respond to via an I3C Target Transaction Interface (if supported and enabled).

For Direct CCCs that the Secondary Controller Logic does not support, or has not been programmed to respond to, the Secondary Controller Logic shall not acknowledge its address (i.e., equivalent to passive NACK in SDR Mode). If the Host Controller supports an I3C Target Transaction Interface to software, and if this interface is also enabled, then the Secondary Controller Logic shall pass notification of that Direct CCC to the Target Transaction Interface for handling. Otherwise, the Host Controller shall report this as an unhandled Direct CCC by reporting the **CCC\_UNHANDLED\_NACK\_STAT** interrupt condition to the Driver.

#### For Broadcast CCCs

The Secondary Controller Logic shall track the CCC modality for the I3C Mode, and shall only capture and act on the Broadcast data messages for those CCCs that it supports internally. If the Host Controller supports the I3C Target Transaction Interface to software, and if this interface is also enabled, then any other Broadcast CCCs shall be forwarded to this Target Transaction Interface. Otherwise, any other Broadcast CCCs shall be discarded without notification.

However, the **RSTDAA** CCC is a special case: if the Secondary Controller Logic does not support the **RSTDAA** CCC (i.e., because it does not support any of the CCCs used for Dynamic Address Assignment), then it shall treat this Broadcast CCC as a fatal error if it receives such a CCC from the Active Controller of the I3C Bus. Such a condition typically applies to a Host Controller implementation that closely aligns with configuration Option 1 or Option 2 (per *Section 6.17.1*); and does not support configuration Option 3, where the Host Controller joins the I3C Bus as a Secondary Controller and typically receives a Dynamic Address from another Controller-capable Device on the I3C Bus. However, in most cases the Secondary Controller Logic should support the **RSTDAA** CCC.

#### Note:

*If the **RSTDAA** CCC is received but the Secondary Controller cannot successfully handle it (i.e., because it does not support Dynamic Address Assignment) then the Host Controller shall report this situation as an error by reporting the **CCC\_FATAL\_RSTDAA\_ERR\_STAT** interrupt condition, to signal that the Host Controller cannot continue to operate without intervention from the Driver.*

### 6.17.3.1.1 CCCs Serviced by Standard Secondary Controller Logic

The standard Secondary Controller Logic shall support autonomous response to the following CCCs:

#### Minimum Required CCCs for Standby Controller Mode:

- **GETACCCR** (see *Section 6.17.4*).

Readiness to accept the **GETACCCR** CCC is determined by the value of field **ACR\_FSM\_OP\_SELECT** in register **STBY\_CR\_CONTROL** (*Section 7.7.11.1*), and by additional logic within the Host Controller that determines readiness to accept this CCC and participate in the Controller Role Handoff Procedure (see *Section 6.17.4*).

- **GETSTATUS** Format 1 (i.e., without a Defining Byte), and Format 2 with Defining Byte value **PRECR** (i.e., 0x91) shall be supported.

A minimal Standby Controller implementation shall support this CCC, and shall either:

(A) Omit any configuration register that would allow the Driver to provide custom values for the CCC response data (in this case, responses shall use appropriate values for mandatory fields); or

(B) Implement a vendor-defined configuration register (i.e., in Extended Capabilities) that allows the Driver to provide custom values for the CCC response data (in this case, responses shall use values from this register's fields).

- **GETCAPS**

These CCCs shall be handled automatically via register **STBY\_CR\_CCC\_CONFIG\_GETCAPS** (*Section 7.7.11.10*), and the responses shall indicate the capabilities of the Secondary Controller Logic. Since the Secondary Controller Logic (as defined in this I3C HCI Specification) only supports a subset of the possible I3C Target role's capabilities, this register only exposes the capabilities that are supported.

- **ENECA** and **DISEC**

A minimal Standby Controller implementation shall support the **ENECA** and **DISEC** CCCs, with the **ENCR** and **DISCR** bits. However, a configuration register is not required for this use case.

Note that support for additional event notification bits that are defined for these CCCs may be added at the discretion of the implementer, via Extended Capabilities.

The Secondary Controller Logic may also support autonomous response to the following CCCs:

**4940      Optional CCCs for Standby Controller Mode:**

- 4941      • ENTDAA, RSTDAA, SETDASA, SETAASA, and SETNEWDA**

4942      These CCCs shall be handled automatically via register **STBY\_CR\_DEVICE\_ADDR**  
4943      (*Section 7.7.11.2*). For the **ENTDAA** CCC, see also registers **STBY\_CR\_DEVICE\_CHAR**  
4944      (*Section 7.7.11.5*) and **STBY\_CR\_DEVICE\_PID\_LO** (*Section 7.7.11.6*).

4945      Note that Standby Controller support for each of the Dynamic Address Assignment CCCs  
4946      (**ENTDAA**, **SETDASA**, and **SETAASA**) can be enabled or disabled by a bitfield in register  
4947      **STBY\_CR\_CONTROL** (*Section 7.7.11.1*): **DAA\_ENTDAA\_ENABLE**, **DAA\_SETDASA\_ENABLE**,  
4948      and **DAA\_SETAASA\_ENABLE**, respectively. The Driver shall enable or disable response for  
4949      each of these CCCs, based on methods that might be implemented and supported in  
4950      corresponding fields in register **STBY\_CR\_CAPABILITIES** (*Section 7.7.11.3*).

- 4951      • GETPID, GETBCR, and GETDCR**

4952      These shall be handled automatically via registers **STBY\_CR\_DEVICE\_CHAR**  
4953      (*Section 7.7.11.5*) and **STBY\_CR\_DEVICE\_PID\_LO** (*Section 7.7.11.6*).

4954      Note that support for some or all of these CCCs shall be conditionally required,  
4955      depending on which CCCs for Dynamic Address Assignment are also supported. For  
4956      example, if the **ENTDAA** CCC is supported, then all of these CCCs must also be  
4957      supported.

- 4958      • RSTACT**

4959      These CCCs shall be handled automatically via register  
4960      **STBY\_CR\_CCC\_CONFIG\_RSTACT\_PARAMS** (*Section 7.7.11.11*).

4961      Additionally, the Secondary Controller Logic shall handle defined Target Reset action  
4962      flows with the **RSTACT** CCC followed by the Target Reset Pattern, according to the  
4963      configuration of registers **STBY\_CR\_CCC\_CONFIG\_RSTACT\_PARAMS** and  
4964      **STBY\_CR\_CONTROL** (*Section 7.7.11.1*). If the Active Controller sends a Target Reset  
4965      action to reset the Secondary Controller Logic, then the Host Controller shall, at the  
4966      discretion of the implementer, either assert interrupt **STBY\_CR\_OP\_RSTACT\_STAT**  
4967      (*Section 7.7.11.7*), or else reset the Host Controller internally.

- 4968      • DEFTGTS and DEFGRPA**

4969      These Broadcast CCCs shall be handled automatically and enqueued as Read data into  
4970      the IBI Queue/Ring (see *Section 6.17.3.2*).

### 6.17.3.1.2 CCCs Serviced by Extensions to Secondary Controller Logic

4971      The Host Controller's Secondary Controller Logic may support additional CCCs that are defined in the I3C  
4972      Specification, implemented as standard extensions to Secondary Controller Logic that might be required for  
4973      certain use cases or optional features of an I3C Device. Such extensions shall be exposed as instances of a  
4974      vendor-specific Extended Capability structure that might be defined to control such CCC handling.

### 6.17.3.1.3 CCCs Serviced by Other Mechanisms

4975      Any CCCs (either Direct or Broadcast) that are not serviced autonomously by the Secondary Controller  
4976      Logic shall either be treated as variants of either Read-Type or Write-Type transactions, and would  
4977      typically be handled by software via the I3C Target Transaction Interface, or else treated as extensions to  
4978      the Secondary Controller Logic that support autonomous handling of custom CCCs, with configuration  
4979      registers exposed via one or more instances of a vendor-defined Extended Capability structure indicating  
4980      custom CCC handling.

#### 6.17.3.1.4 Implications for Minimal Standby Controller Use Case (Informative)

In addition to *Section 6.17.3.1.1*'s list of CCCs that a minimal Standby Controller implementation is required to support, it is expected that such a minimal implementation will usually be the Active Controller of the I3C Bus, and will be responsible for most Bus Configuration functions. For many such use cases, such minimal Host Controllers are typically paired with other limited-function Secondary Controller Devices that are expected to only perform limited tasks while serving as the Active Controller, and only to hold the Controller Role for a limited time.

Such other Secondary Controller Devices would be expected to support only the minimum I3C Bus Management procedures while acting as Active Controller (see the I3C Specification [*MIPI02*] at *Section 5.1.7.3.2*), and to not attempt to perform additional Bus Configuration procedures while acting as Active Controller. Since the Host Controller would only support the minimal Standby Controller mode functions and capabilities, and would only pass the Controller Role to such Secondary Controller Devices for a limited duration and purpose, this system configuration should present few integration issues for the Host Controller.

It is also clear that such limited-purpose Secondary Controller Devices should typically pass the Controller Role back to the Host Controller when finished with the task for which the Controller Role was initially transferred. Additionally, such Secondary Controller Devices should support sending the Controller Role Request to initiate the process of handing the Controller Role back to the Host Controller (see *Section 6.17.3.3*). The I3C Specification recommends that such Secondary Controller Devices should advertise such capabilities using the **GETCAPS** Format 2 CCC with Defining Byte value **CRCAPS** (i.e., 0x91), which includes a field for the automatic pass-back capability (see the I3C Specification [*MIPI02*] at *Section 5.1.9.3.19*).

However, if a more fully-featured Secondary Controller Device were to be paired with a minimal Standby Controller implementation of a Host Controller, then issues could arise due to a mismatch of expectations. Several error scenarios are possible:

- The Secondary Controller Device attempting to perform Bus Configuration procedures, which the Host Controller would not be able to successfully track or understand due to its minimal capabilities.
- The Secondary Controller Device attempting to send certain CCCs or other Private Read/Write transactions to the Host Controller, with the expectation of a particular response that would not be possible due to the Host Controller's limited capabilities, or its lack of support for other CCCs.
- The Secondary Controller Device attempting to assign a new Dynamic Address to the Host Controller, or to reset all Dynamic Addresses on the Bus via the **RSTDAA** CCC, which the Host Controller would not support (and might cause a fatal error, per *Section 6.17.3.1*).

System designers should ensure that minimal Standby Controller implementations of Host Controllers are used for system integrations where other Secondary Controller Devices have carefully prescribed roles and capabilities, and where such other Devices will be known (or configured) to not exceed the boundaries of what the minimal Host Controller can support in Standby Controller mode. For use cases where that is not possible or where those limits cannot be ensured, a more fully-featured Host Controller with Standby Controller mode support is strongly recommended.

#### 6.17.3.2 Receiving Broadcast DEFTGTS and DEFGRPA CCCs

A Host Controller that supports Standby Controller mode shall monitor the I3C Bus for specific Broadcast CCCs relating to transferring Bus configuration (i.e., **DEFTGTS** and **DEFGRPA**) that can be sent by the Active Controller. The Secondary Controller Logic shall receive such Broadcast CCCs while enabled (i.e., while field **STBY\_CR\_ENABLE\_INIT** in register **STBY\_CR\_CONTROL** has a value of 2'b10, per *Section 7.7.11.1*).

5024

**Note:**5025  
5026  
5027

If field **STBY\_CR\_ENABLE\_INIT** in register **STBY\_CR\_CONTROL** does not have a value of 2'b10, then either the Secondary Controller Logic is disabled, or it will not receive and capture Broadcast CCCs.

5028  
5029  
5030

Any time the Active Controller sends either of these Broadcast CCCs (i.e., **DEFTGTS** or **DEFGRPA**), the Secondary Controller Logic shall capture each CCC and its subsequent data bytes, and make them available in the active IBI Data Queue or IBI Ring Pair, depending on the current operating mode:

5031

- **In PIO Mode:** The active IBI Data Queue is accessed via register **IBI\_PORT** (*Section 7.5.4*).

5032  
5033  
5034  
5035  
5036

- The Secondary Controller Logic shall make the Broadcast CCC data available as a sequence of one or more IBI Status Descriptors, with each followed by IBI Data DWORDs according to the standard IBI handling defined in *Section 6.9.1*. The first IBI Data DWORD after the first such IBI Status Descriptor shall contain the Broadcast Address (7'h7E) in the first byte, the Command Code in the second byte, and subsequent bytes afterwards.

5037  
5038  
5039  
5040  
5041

- As necessary, the Secondary Controller Logic shall flow the data into multiple reports, each with its own IBI Status Descriptor, per the threshold setting in field **IBI\_DATA\_SEGMENT\_SIZE** in register **QUEUE\_THLD\_CTRL** (*Section 7.5.5*). Field **LAST\_STATUS** shall be set to 1'b0 for every IBI Status Descriptor except the last IBI Status Descriptor, where field **LAST\_STATUS** shall be set to 1'b1 (i.e., similar to a regular IBI with a split payload).

5042  
5043

- **IN DMA Mode:** The active IBI Ring Pair used is selected by field **BCAST\_CCC\_IBI\_RING** in register **STBY\_CR\_CONTROL** (*Section 7.7.11.1*).

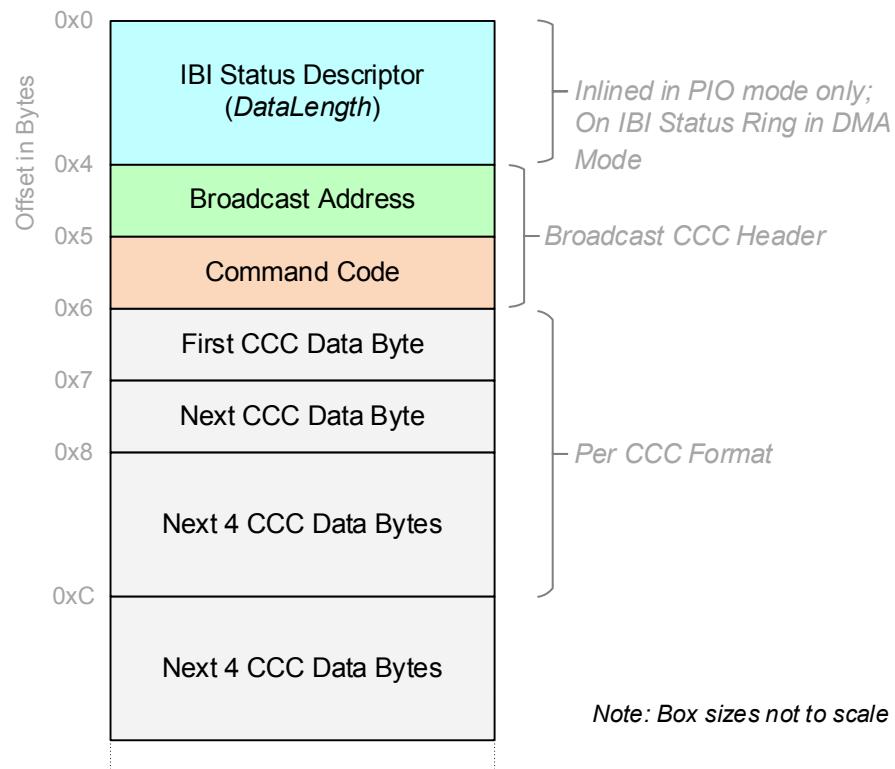
5044  
5045  
5046  
5047  
5048  
5049  
5050  
5051  
5052  
5053  
5054

- The selected Ring Bundle must be enabled and configured for use, and its IBI Ring Pair must also be initialized and ready to receive data per *Section 6.6.1*.
- The Secondary Controller Logic shall make the Broadcast CCC data available as a sequence of one or more IBI Status Descriptors in the IBI Status Ring, with appropriately allocated chunks of data in the IBI Data Ring per *Section 6.6.3*. The first IBI Data DWORD shall contain the Broadcast Address (7'h7E) in the first byte, the Command Code in the second byte, and subsequent bytes afterwards.
- As necessary, the Secondary Controller Logic shall flow the data into multiple reports, each with its own IBI Status Descriptor and set of Data Chunks, per the value of field **CHUNK\_SIZE** in register **IBI\_SETUP** (see *Section 7.6.10.2*). Field **LAST\_STATUS** shall be set to 1'b0 for every IBI Status Descriptor except the last IBI Status Descriptor, where field **LAST\_STATUS** shall be set to 1'b1 (i.e., similar to a regular IBI with a split payload).

5055  
5056

Each IBI Status Descriptor for Broadcast CCC data shall have field **STATUS\_TYPE** set to 3'b111 to distinguish it from other IBI Status Descriptor uses (see *Section 8.6.7*).

5057  
5058  
5059 **Figure 37** illustrates the data format of the IBI Status Descriptor and the IBI Data DWORDs that the Host Controller will generate for each Broadcast **DEFTGTS** and **DEFGRPA** CCC that it receives from the Active Controller.



5060 **Figure 37 Secondary Controller Broadcast CCC Data Format for IBI Event**

5061 **Note:**

5062 Implementers may choose to support capturing additional Broadcast CCCs in addition to the  
5063 **DEFTGTS** and **DEFGRPA** CCCs. If so, then each such Broadcast CCC shall be captured using the  
5064 same data format (see **Figure 37**).

5065 If a Broadcast CCC is sent with a Defining Byte (per the I3C Specification [**MIPI02**] at  
5066 **Section 5.1.9.1**) then the Defining Byte will appear as the first CCC data byte after the Command  
5067 Code (i.e., at byte offset 0x6).

5068 In DMA Mode byte offsets 0x0 through 0x3 will be the IBI Status Descriptor on the IBI Status Ring,  
5069 and byte offsets 0x4 and above will be in one or more data chunks on the IBI Data Ring (i.e.,  
5070 without the initial 4-bytes IBI Status Descriptor shown in **Figure 37**).

### 6.17.3.3 Simple Controller Role Request

A Host Controller that supports the Minimal Standby Controller capabilities should also support the simple method of sending a Controller Role Request to the Active Controller of the I3C Bus defined in this section. This method involves reading from and writing to specific fields in the registers defined in **Section 7.7.11**. If the Host Controller supports this Simple Controller Role Request flow, then it shall indicate this capability with a value of 1'b1 in field **SIMPLE\_CRR\_SUPPORT** of register **STBY\_CR\_CAPABILITIES** (**Section 7.7.11.3**).

**Note:**

*Although a Minimal Standby Controller (as defined in this I3C HCI Specification) is not required to support this flow, it is strongly recommended.*

The Controller Role Request shall be initiated when the Driver writes to field **CR\_REQUEST\_SEND** in register **STBY\_CR\_CONTROL** (**Section 7.7.11.1**). Once the Driver writes a value of 1'b1 to this field, the Host Controller shall capture the request. After capturing the request, the Host Controller shall indicate the request status in field **SIMPLE\_CRR\_STATUS** in register **STBY\_CR\_STATUS** (see **Section 7.7.11.4**).

**Note:**

*Field **CR\_REQUEST\_SEND** is a write-only bit. A pending Controller Role Request cannot be cancelled by the Driver, and the Host Controller shall only allow one pending request to be enqueued at a time. Writes to field **CR\_REQUEST\_SEND** shall have no effect if field **SIMPLE\_CRR\_STATUS** in register **STBY\_CR\_STATUS** currently has a value of 3'b001.*

The Secondary Controller Logic shall attempt to send the Controller Role Request at the earliest opportunity (i.e., after waiting for the I3C Bus to be idle for at least the Bus Available Condition, per the I3C Specification [**MIPI02**] at **Section 5.1.3.2.2**). The Driver may check on the status of the Controller Role Request by reading field **SIMPLE\_CRR\_STATUS** in register **STBY\_CR\_STATUS** (**Section 7.7.11.4**). The returned value indicates the status of the Secondary Controller Logic for this pending request, and should be used by the Driver to understand whether the Controller Role Request has been sent, and if so, whether the Active Controller ACKed or NACKed the Controller Role Request.

When the Active Controller responds to this Controller Role Request, the Host Controller shall also trigger the **CRR\_RESPONSE\_STAT** interrupt in register **STBY\_CR\_INTR\_STATUS** (**Section 7.7.11.7**). When this interrupt is triggered, field **SIMPLE\_CRR\_STATUS** shall have a valid value of 3'b010 or higher.

**Note:**

*If a Controller Role Request was NACKed, then the Driver should use the value of field **SIMPLE\_CRR\_STATUS** to determine whether to send a subsequent Controller Role Request, for cases in which it is appropriate to do so. The Host Controller shall not prevent the Driver from sending subsequent Controller Role Requests after receiving a NACKed Controller Role Request, which might be followed by the **DISEC** CCC with the **DISCR** bit (i.e., the Active Controller disabling Controller Role Requests on the I3C Bus).*

*Even if the Controller Role Request was ACKed, that is no guarantee that the Active Controller will always use the **GETACCCR** CCC to attempt to pass the Controller Role. The I3C Specification states that passing the Controller Role may happen either immediately, or at a later time. Additionally, the Active Controller might need to perform additional steps to synchronize this Host Controller's Secondary Controller Logic with the current state of the I3C Bus and all known I3C Devices. In order to remain ready, the Host Controller should be primed to accept the Controller Role per **Section 6.17.4** before the Driver sends the Controller Role Request.*

#### 6.17.4 Transition from Standby Controller Mode to Active Controller Mode

In order to start the transition to Active Controller mode, the Host Controller in Standby Controller mode shall prepare the Secondary Controller Logic to accept the anticipated **GETACCCR** CCC from the Active Controller.

The Host Controller becomes ready to accept the **GETACCCR** CCC according to the conditions below, using either of these two flows:

- **Direct Prime After Transition:** If field **ACR\_FSM\_OP\_SELECT** in register **STBY\_CR\_CONTROL** (*Section 7.7.11.1*) was previously set to 1'b0, and the Driver subsequently writes the value 1'b1 to field **ACR\_FSM\_OP\_SELECT**, then the Secondary Controller Logic shall become primed to accept the Controller Role.
- **Auto-Prime With Transition:** The Secondary Controller Logic might have been previously primed to automatically accept the Controller Role upon transitioning into Standby Controller mode, if field **PRIME\_ACCEPT\_GETACCCR** was set to 1'b1 when the Host Controller accepted the special Command Descriptor in Active Controller Mode (per *Section 6.17.2*) and then successfully transitioned into Standby Controller mode.

**Note:**

*The Driver may choose to initially follow the Auto Prime with Transition path, then write 1'b0 to field **ACR\_FSM\_OP\_SELECT** to remove the primed status; and then subsequently write 1'b1 to field **ACR\_FSM\_OP\_SELECT** to prime the Secondary Controller Logic again, at its discretion. However, for a limited-scope Host Controller (i.e., Minimal Secondary Controller) implementation, field **ACR\_FSM\_OP\_SELECT** might not be writeable in Standby Controller mode.*

The Driver may explicitly request the Controller Role from the Active Controller by instructing the Secondary Controller Logic to send the Controller Role Request via the Simple Controller Role Request mechanism specified in *Section 6.17.3.3*. This may occur either before or after writing 1'b1 to field **ACR\_FSM\_OP\_SELECT**; however, the typical flow expects the Secondary Controller Logic to be primed (i.e., ready to accept the **GETACCCR** CCC) before it sends the Controller Role Request.

**Note:**

*Sending a Controller Role Request is not required to meet the conditions for readiness to accept the **GETACCCR** CCC. For certain use cases, the Active Controller might be configured to automatically send the **GETACCCR** CCC without having received a Controller Role Request. If a Controller Role Request is not sent, then the Secondary Controller Logic must wait until the Active Controller sends the **GETACCCR** CCC at its own discretion.*

If the Host Controller supports a Deep Sleep state for power management, then it shall automatically write a value of 1'b1 to field **HANOFF\_DEEP\_SLEEP** in register **STBY\_CR\_CONTROL** (*Section 7.7.11.1*) to signal to the Active Controller that it might have missed important Broadcast messages (such as **DEFTGTS** and **DEFGRPA** CCCs) while in that state. This provides indication that the Active Controller should re-send Bus Configuration Broadcast CCCs, if such changes were recently made. Alternatively, software may choose to report the same Deep Sleep indication by writing to field **HANOFF\_DEEP\_SLEEP**.

The readiness to accept the **GETACCCR** CCC depends on several conditions, all of which must remain true:

1. The Secondary Controller Logic has been initialized and configured as defined in *Section 6.17.1.1*.  
This includes setting field **STBY\_CR\_ENABLE\_INIT** in register **STBY\_CR\_CONTROL** (*Section 7.7.11.1*) to a non-zero value. Note that this is supported for either initial role on the I3C Bus, so long as the Secondary Controller Logic is eventually enabled.
2. The Bus Controller Logic has been configured.  
This includes setting field **BUS\_ENABLE** in register **HC\_CONTROL** (*Section 7.4.2*).
3. The Host Controller has been configured with a valid Dynamic Address.

5159 In register **STBY\_CR\_DEVICE\_ADDR** (*Section 7.7.11.2*), field **DYNAMIC\_ADDR\_VALID** must  
5160 be set to 1'b1 and field **DYNAMIC\_ADDR** must be set to a Dynamic Address that is valid for  
5161 an I3C Device.

5162 **Note:**

5163 *If the Secondary Controller Logic receives a successful Target Reset action (i.e., **RSTACT**  
5164 **CCC** followed by the Target Reset Pattern) that resets the Dynamic Address, as configured  
5165 in register **STBY\_CR\_CCC\_CONFIG\_RSTACT\_PARAMS** (*Section 7.7.11.11*), then the  
5166 Secondary Controller Logic is required to revoke readiness to accept the Controller Role.*

- 5167 4. The value 1'b1 has been written to field **ACR\_FSM\_OP\_SELECT** in register **STBY\_CR\_CONTROL**  
5168 (*Section 7.7.11.1*) to signal readiness to accept the **GETACCCR** CCC.  
5169 This sets the vector for the intended transition direction (i.e., the role that the Driver  
5170 wishes the Host Controller to take). Software may read this register to verify that field  
5171 **ACR\_FSM\_OP\_SELECT** contains 1'b1.  
5172 5. The Device has not received any incoming Broadcast **DEFTGTS** or **DEFGRPA** CCCs that have not yet  
5173 been processed by software and had appropriate status cleared to indicate acceptance of the message  
5174 data in these Broadcast CCCs.  
5175 A. Field **HANOFF\_DELAY\_NACK** in register **STBY\_CR\_CONTROL** (*Section 7.7.11.1*) indicates whether  
5176 these Broadcast CCCs have been received by Secondary Controller Logic but not yet processed by  
5177 software. Field **HANOFF\_DELAY\_NACK** shall contain 1'b1 while that condition is false (i.e., is not  
5178 met).  
5179 • If the Secondary Controller Logic receives either of these Broadcast CCCs, then it shall  
5180 automatically clear field **HANOFF\_DEEP\_SLEEP** in register **STBY\_CR\_CONTROL**  
5181 (*Section 7.7.11.1*) if that field was previously set, per *Section 6.17.4.2*.  
5182 In this case, the Host Controller will capture the Broadcast CCC data and make  
5183 it available to the Host via the active IBI Queue/Ring (per *Section 6.17.3.2*).  
5184 • The Secondary Controller Logic shall not use the value of field **HANOFF\_DEEP\_SLEEP** to  
5185 determine whether to accept the Controller Role (i.e., the **GETACCCR** CCC).  
5186 B. Software must clear this status by writing the value 1'b1 to field **HANOFF\_DELAY\_NACK** (i.e.,  
5187 write 1 to clear) in order to make this condition true. Software should not clear this condition until  
5188 it has handled the Broadcast CCCs appropriately, per *Section 6.17.4.1*.  
5189 • If the I3C Target Transaction Interface is supported and enabled, then software should verify  
5190 that all received Request Descriptors for such Broadcast CCCs have been processed. Software  
5191 might need to check this field again, in case the Active Controller subsequently sends any  
5192 Broadcast **DEFTGTS** or **DEFGRPA** CCCs again.  
5193 • If the I3C Target Transaction Interface is not supported or not enabled, then some other  
5194 extended Secondary Controller Logic is required to handle the data messages in such  
5195 Broadcast CCCs, and then clear this condition automatically (i.e., without software  
5196 involvement).  
5197 C. If the Active Controller repeatedly sends such Broadcast CCCs while software attempts to process  
5198 them as they are received, then such activity might prevent this condition from remaining true, and  
5199 this would hinder the Secondary Controller Logic from accepting the **GETACCCR** CCC. Note that  
5200 the Host Controller in Standby Controller mode has no control over the Active Controller's ability  
5201 to continue sending such Broadcast CCCs.  
5202 6. If an I3C Target Transaction Interface is supported and enabled, then the Device has not received any  
5203 other incoming Write-Type transactions (either as Private Writes, Broadcast CCCs, or Direct CCCs)  
5204 that have not yet been processed by software.  
5205 A. This does not include the Broadcast **DEFTGTS** or **DEFGRPA** CCCs that are required to be  
5206 captured by Secondary Controller Logic (as listed above, and in *Section 6.17.3.2*), although the  
5207 method of handling them is similar. Field **PENDING\_RX\_NACK** in register **STBY\_CR\_CONTROL**  
5208 (*Section 7.7.11.1*) indicates whether these transactions have been received by Secondary  
5209 Controller Logic but not yet processed by software.

- 5210        B. Software must clear this status by writing the value 1'b1 to field **PENDING\_RX\_NACK** (i.e., write 1  
5211        to clear) in order to make this condition true, per **Section 6.17.4.1**.

5212        **Note:**

5213        *If an I3C Target Transaction Interface is not enabled (or is not supported), then such Write-Type  
5214        transactions will not be received by software, and condition 6 above does not apply.*

5215        Once all of the above conditions are true, for and as long as they remain true, the Secondary Controller  
5216        Logic shall automatically accept the **GETACCCR** CCC once the Active Controller sends it, directed to this  
5217        Host Controller's Dynamic Address. If the Secondary Controller Logic accepts the **GETACCCR** CCC, then  
5218        it shall return its own Dynamic Address and the correct Parity Bit at the appropriate time in the CCC  
5219        framing (see the I3C Specification [**MIPI02**] at **Section 5.1.9.3.16**).

5220        However, if any of the above conditions are not met, then the Secondary Controller Logic either will not  
5221        receive the **GETACCCR** CCC (i.e., because it is not enabled), or else shall not accept the **GETACCCR** CCC,  
5222        per **Section 6.17.4.1**.

5223        **Note:**

5224        *Automatic acceptance of the **GETACCCR** CCC does not strictly require the I3C Target Transaction  
5225        Interface to software for Read-Type or Write-Type transactions. However, successfully processing  
5226        the Broadcast CCCs (such as **DEFTGTS** and **DEFGRPA**) requires either the I3C Target  
5227        Transaction Interface to be supported and enabled, or an IBI Queue/Ring (per operating mode) to  
5228        cache the data messages from such Broadcast CCCs and make them available to software, either  
5229        before or after the transition to Active Controller mode.*

5230        *If the I3C Target Transaction Interface is supported and enabled, then such Broadcast CCCs shall  
5231        appear as Write-Type transactions if sent by the Active Controller. Software should read these  
5232        transactions according to the operating mode, and handle them in an appropriate manner.*

5233        Following acceptance of the **GETACCCR** CCC, as indicated by the Active Controller driving the STOP  
5234        condition, the Secondary Controller Logic shall participate in the Controller-to-Controller Handoff  
5235        Procedure (see the I3C Specification [**MIPI02**] at **Section 5.1.7.2**), activate the Bus Controller Logic (i.e.,  
5236        enable SCL clock generation), and assert its Controller Role within the appropriate time, to prevent losing  
5237        the Controller Role due to Error Type CE3 detection by the former Active Controller.

5238        If all of these events occur as indicated, then the Host Controller shall become the Active Controller of the  
5239        I3C Bus, and shall transition into Active Controller mode. A successful transition shall trigger the  
5240        **STBY\_CR\_ACCEPT\_OK\_STAT** interrupt, signaling that the Host Controller is now operating in Active  
5241        Controller mode and that the Secondary Controller Logic is now idle.

5242        If there is an error during the **GETACCCR** CCC, or if the Active Controller does not indicate acceptance  
5243        (i.e., by driving the STOP condition), then this shall trigger the **STBY\_CR\_ACCEPT\_ERR\_STAT** interrupt,  
5244        signaling that the Host Controller is still operating in Standby Controller mode.

5245        As and when the Active Controller sends the **GETACCCR** CCC to this Host Controller's Dynamic Address,  
5246        and as the Secondary Controller Logic responds to that CCC, the Host Controller shall trigger one or more  
5247        of the **Table 10** interrupt events in register **STBY\_CR\_INTR\_STATUS** (**Section 7.7.11.7**) to report successful  
5248        transitions or failed transition attempts from Standby Controller mode to Active Controller mode.

5249  
5250**Table 10 Possible Interrupt Events Reported for Accepting or Not Accepting Controller Role**

Interrupt Event	Description
<b>STBY_CR_ACCEPT_NACKED_STAT</b>	The Controller Role was not accepted (i.e., the Secondary Controller Logic NACKed the <b>GETACCCR</b> CCC) because the readiness conditions were not met (per <b>Section 6.17.4.1</b> ).
<b>STBY_CR_ACCEPT_OK_STAT</b>	The Controller Role was accepted (i.e., the Secondary Controller Logic ACKed the <b>GETACCCR</b> CCC) and the Active Controller completed the Controller Role Handoff Procedure successfully, ending the <b>GETACCCR</b> with the STOP condition. The Host Controller is now operating in Active Controller mode, and the I3C Bus Controller Logic is now active.
<b>STBY_CR_ACCEPT_ERR_STAT</b>	The Controller Role was accepted (i.e., the Secondary Controller Logic ACKed the <b>GETACCCR</b> CCC), but the Active Controller did not complete the Controller Role Handoff Procedure successfully, or did not end the <b>GETACCCR</b> CCC with the STOP condition. The Host Controller is still operating in Standby Controller mode.

5251

**Note:**5252  
5253  
5254

The **Table 10** interrupt events describe attempted transitions from Standby Controller mode to Active Controller mode. By contrast, the interrupt events in **Table 9** describe attempted transitions from Active Controller mode to Standby Controller mode.

5255  
5256  
5257  
5258

If the interrupt event **STBY\_CR\_ACCEPT\_ERR\_STAT** is triggered for a failed or cancelled attempt, then this event shall not change the Secondary Controller Logic's primed state, and the Secondary Controller Logic shall remain ready to accept any subsequent **GETACCCR** CCC that the Active Controller might send.

5259  
5260  
5261  
5262  
5263  
5264  
5265  
5266  
5267  
5268

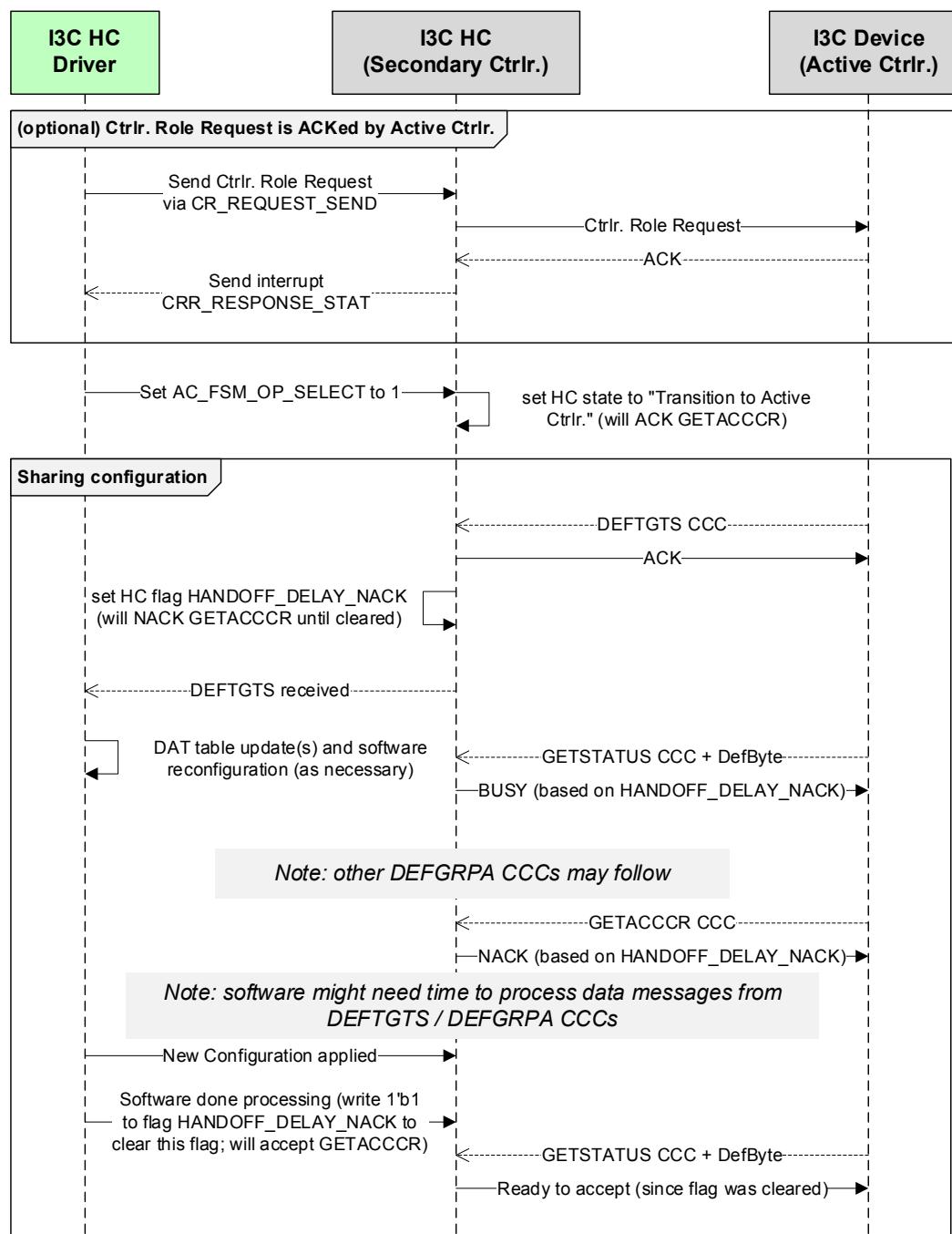
If two or more of these interrupt events are triggered, then this reflects that the Active Controller has made multiple attempts to pass the Controller Role, and that these attempts ended in different outcomes. Since multiple events can be indicated by this register, if event **STBY\_CR\_ACCEPT\_OK\_STAT** plus any of the other events are triggered, and if field **AC\_CURRENT\_OWN** in register **STBY\_CR\_STATUS** contains the value 1'b1, then the successful transition was the most recent attempt, and the Host Controller has finally transitioned into Active Controller mode. To prevent past interrupt events from being confused with subsequent transitions to (and then from) Standby Controller mode, the software should clear any triggered interrupt events listed in **Table 10** after it determines that the Host Controller has successfully transitioned into Active Controller mode.

5269  
5270  
5271  
5272  
5273

Software may monitor the transition status by reading field **AC\_CURRENT\_OWN** in register **STBY\_CR\_STATUS** (**Section 7.7.11.4**). If this field still contains the value 1'b0, then the Host Controller is still operating in Standby Controller mode. As long as a successful transition has not yet occurred, software may prevent a future transition (i.e., may revoke readiness to accept the **GETACCCR** CCC) by writing the value 1'b0 to field **ACR\_FSM\_OP\_SELECT** in register **STBY\_CR\_CONTROL** (**Section 7.7.11.1**).

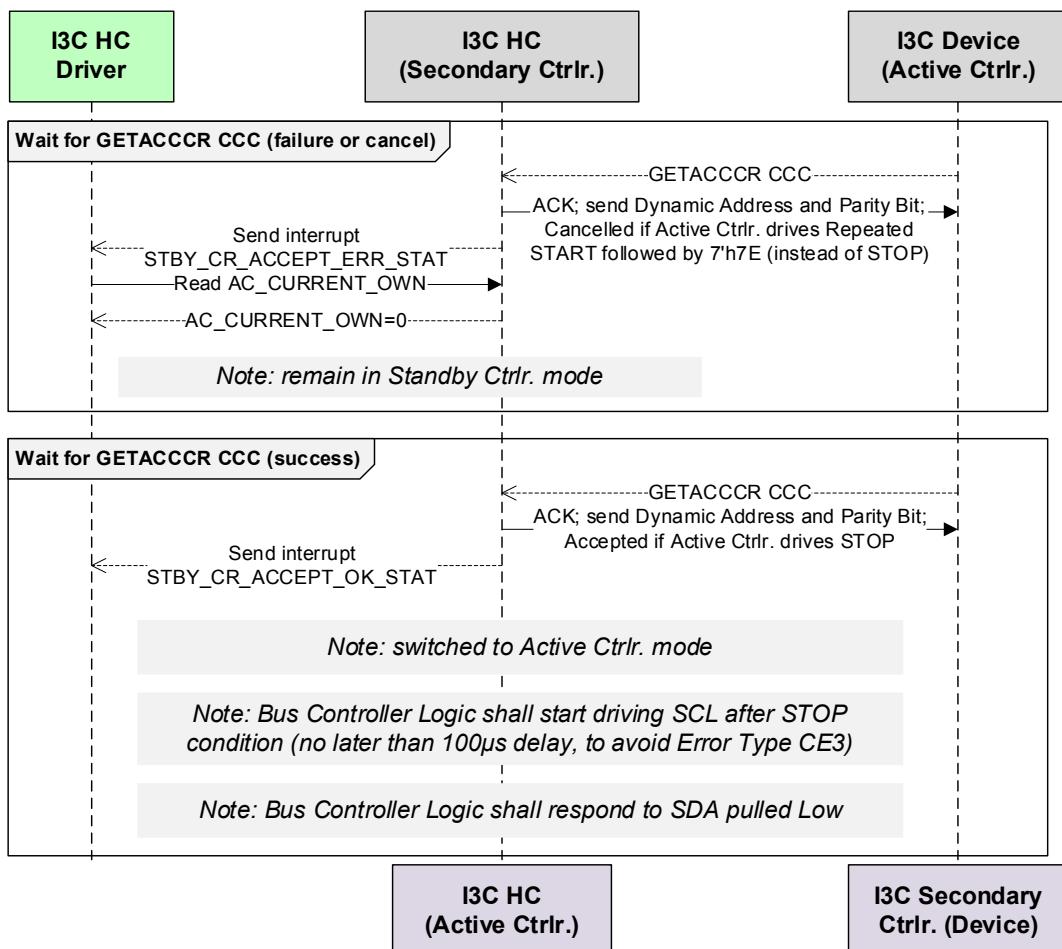
5274

**Figure 38** and **Figure 39** together illustrate the flow for transitioning to Active Controller mode.



**Figure 38 Flow for Transition to Active Controller Mode, Part 1**

5275



5276

**Figure 39 Flow for Transition to Active Controller Mode, Part 2**

#### 6.17.4.1 Conditions for Not Accepting the GETACCCR CCC

5277 If field **ACR\_FSM\_OP\_SELECT** in register **STBY\_CR\_CONTROL** (*Section 7.7.11.1*) is set to 1'b0, and the  
 5278 Active Controller sends the **GETACCCR** CCC, then the Host Controller shall not accept (i.e., shall NACK)  
 5279 the CCC.

5280 This can happen under any or all of the following conditions:

- 5281 • The Host Controller enters Standby Controller mode, and was not initially primed to automatically accept  
 5282 the Controller Role (i.e., field **PRIME\_ACCEPT\_GETACCCR** was set to 1'b0);  
 5283 • The Host Controller initializes in Standby Controller mode with field **ACR\_FSM\_OP\_SELECT** set to 1'b0;  
 5284 or  
 5285 • If field **ACR\_FSM\_OP\_SELECT** was set to 1'b1 when the Secondary Controller Logic was primed to accept  
 5286 the Controller Role, but the Driver later wrote 1'b0 to the same field.

5287 **Note:**

5288 As noted in *Section 6.17.4*, these conditions are not completely exclusive: the current value of field  
 5289 **ACR\_FSM\_OP\_SELECT** indicates whether the Secondary Controller Logic is primed to accept the  
 5290 Controller Role, and this status may be changed repeatedly by the Driver at its discretion.

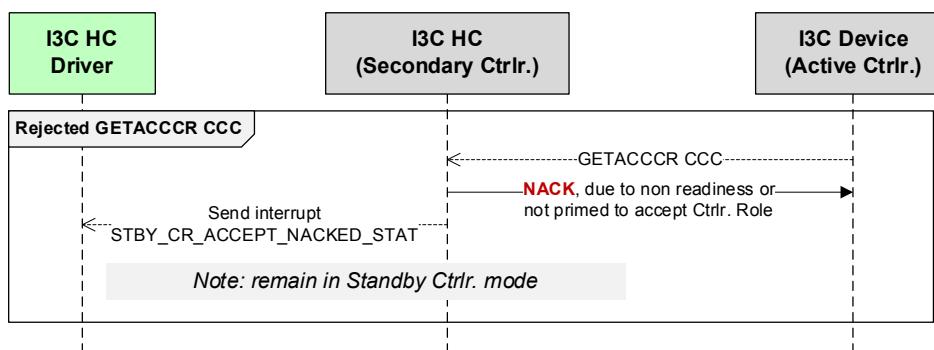
5291 The Secondary Controller Logic shall NACK the **GETACCCR** CCC if the Secondary Controller Logic  
 5292 detects an attempt by the Active Controller to pass the Controller Role to the Host Controller while field  
 5293 **ACR\_FSM\_OP\_SELECT** has a value of 1'b0; while field **HANOFF\_DELAY\_NACK** has a value of 1'b1; or

5294 while field **PENDING\_RX\_NACK** has a value of 1'b1. In any of these events, the Host Controller shall also  
 5295 trigger the **STBY\_CR\_ACCEPT\_NACKED\_STAT** interrupt to signal that the Secondary Controller Logic did  
 5296 not accept the **GETACCCR** CCC that the Active Controller sent, and the Host Controller is still operating in  
 5297 Standby Controller mode.

5298 The Driver may subsequently process this interrupt, and it might decide to instruct the Secondary  
 5299 Controller Logic to accept any subsequent **GETACCCR** CCCs that the Active Controller might send, by  
 5300 priming the Secondary Controller Logic (if not already primed) and resolving any blocking conditions that  
 5301 prevented the NACK of this CCC. Alternately, the Driver may prime the Secondary Controller Logic at any  
 5302 time, and also resolve such blocking conditions proactively (i.e., in advance of receiving the interrupt) if it  
 5303 expects that the Active Controller might send the **GETACCCR** CCC.

5304 In either case, the Driver should take the following steps to resolve whatever blocking conditions might  
 5305 have arisen:

- If field **ACR\_FSM\_OP\_SELECT** was previously set to 1'b0 (as mentioned above), then write a value of 1'b1 to field **ACR\_FSM\_OP\_SELECT**.
- If the blocking condition indicated by field **HANOFF\_DELAY\_NACK** with a value of 1'b1 was in effect, then first process all incoming Broadcast **DEFTGTS** or **DEFGRPA** CCCs per *Section 6.17.4*, and then write 1'b1 (i.e., write 1 to clear) to field **HANOFF\_DELAY\_NACK**.
- If the blocking condition indicated by field **PENDING\_RX\_NACK** with a value of 1'b1 was in effect, then first process or consume all other incoming Write-Type transactions per *Section 6.17.4*, and then write 1'b1 (i.e., write 1 to clear) to field **PENDING\_RX\_NACK**.



5314

**Figure 40 Flow for Rejecting GETACCCR in Standby Controller Mode**

#### 6.17.4.2 Reporting Deep Sleep State Before Handoff

As a part of preparing for the Controller Role Handoff Procedure, the Active Controller should check to see whether a Secondary Controller has reported that it has returned from a Deep Sleep state, and as a result might require resynchronization from the Active Controller (see the I3C Specification [*MIPI02*] at *Section 5.1.7.1* and *Section 5.1.9.3.15*). The Active Controller can check for this status, using the **GETSTATUS** Format 2 CCC with Defining Byte value **PRECR** (i.e., 0x91).

A Host Controller that supports Standby Controller mode shall allow the Deep Sleep state to be reported: either by the Driver, or based on internal logic (i.e., an automatic event triggered by power management state transitions, per *Section 6.1.4.1*). This state can be read or set (written) using field **HANDOFF\_DEEP\_SLEEP** in register **STBY\_CR\_CONTROL** (*Section 7.7.11.1*), subject to the following conditions:

- The Host Controller shall automatically set field **HANDOFF\_DEEP\_SLEEP** to 1'b1 upon returning from any power management state in which there is a possibility that it could have missed any Bus Configuration activity (such as the Broadcast **DEFTGTS** and **DEFGRPA** CCCs) sent by the Active Controller.
- The Driver may choose to set field **HANDOFF\_DEEP\_SLEEP** to 1'b1 to report this condition at any time.

Regardless of the reason, setting field **HANDOFF\_DEEP\_SLEEP** to 1'b1 is ‘sticky’: the field shall remain set until any of the following occurs:

- The Host Controller is reset; or
- The Secondary Controller Logic receives at least one Broadcast **DEFTGTS** CCC (i.e., as the Active Controller attempts to resynchronize with the latest Bus Configuration data), which clears this field automatically; or
- The Secondary Controller Logic accepts the **GETACCCR** CCC (per *Section 6.17.4*) and the Host Controller successfully transitions into Active Controller mode.

The Secondary Controller Logic shall also use the current value of field **HANDOFF\_DEEP\_SLEEP** for automatic responses to the **GETSTATUS** Format 2 CCC with Defining Byte **PRECR**. For example, if field **HANDOFF\_DEEP\_SLEEP** contains 1'b1, then the Secondary Controller Logic shall populate Bit[0] of the LSB for the response of this CCC with 1'b1, indicating “Deep Sleep Detected” (see *Section 5.1.9.3.15* of the I3C Specification [*MIPI02*]).

**Note:**

*The Secondary Controller Logic shall automatically provide the LSB for the response to the **GETSTATUS** Format 2 CCC with Defining Byte **PRECR**. By default, the Secondary Controller Logic shall automatically provide a value of 0x00 for the MSB, because the MSB is reserved for vendor-specific use. If the implementer wishes to configure some other value to be provided for the vendor-specific MSB, then this can be configured using a vendor-specific register (i.e., via Extended Capabilities).*

The Active Controller shall determine whether this Host Controller (acting as a Secondary Controller) needs to be resynchronized. When determining whether to accept or refuse the Controller Role (i.e., whether to ACK or NACK the **GETACCCR** CCC), the Secondary Controller shall not rely on seeing a value of 1'b1 in field **HANDOFF\_DEEP\_SLEEP**. Depending on the use case, it might be natural for the Secondary Controller Logic to accept the Controller Role after returning from Deep Sleep without resynchronization, as the Active Controller would have decided to send the **GETACCCR** CCC without sending any Bus Configuration CCCs (i.e., if no configuration changes needed to be transferred).

**Figure 40** illustrates the flow for rejecting (i.e., providing NACK for) the **GETACCCR** CCC due to lack of readiness.

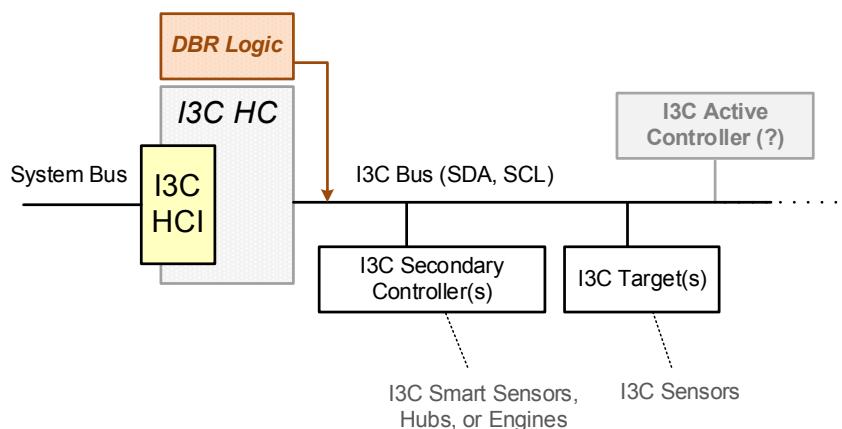
## 6.18 Dead Bus Recovery

The Host Controller may optionally support a Dead Bus Recovery Mechanism that allows the I3C Bus Controller Logic and optional I3C Secondary Controller Logic to either:

- Recover from the Active Controller (i.e., another I3C Device) that might go unresponsive on the I3C Bus, and fail to transfer the Controller Role to this Host Controller;
- Connect to an existing I3C Bus and determine whether there is already an Active Controller that is unresponsive (i.e., momentarily not ready to respond to a Hot-Join Request) or simply not present; or
- Recover from either of situations above, by using the I3C defined Error Type DBR flow (per [Section 5.1.10.1.8](#) of the I3C Specification [[MIP102](#)]) to take control (i.e., to claim the Controller Role) of the I3C Bus.

This mechanism provides additional tools to the Host system, in situations where the reliability and responsiveness of another Active Controller on the I3C Bus cannot be guaranteed, for use cases that rely on this Host system's need to regain control of the I3C Bus in a well-defined manner, according to the Error Type DBR flow. This mechanism also interoperates with optional I3C Secondary Controller Logic for Host Controller implementations that also support Standby Controller mode and can initialize as either the Active Controller or a Secondary Controller (see [Section 6.17.1](#)).

[Figure 41](#) shows the high-level architecture of an I3C Host Controller subsystem that supports the Dead Bus Recovery Mechanism.



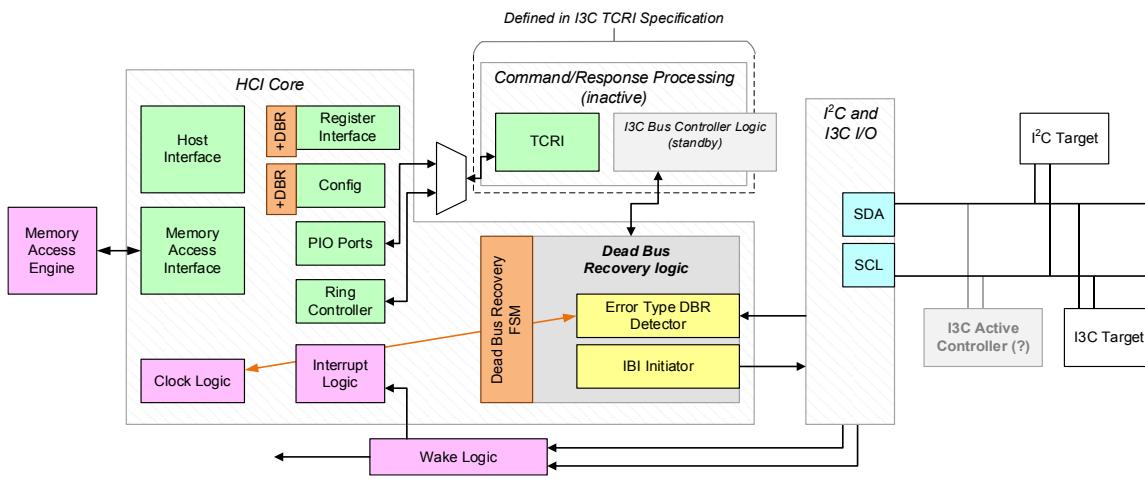
**Figure 41 Example I3C System with Dead Bus Recovery**

A Host Controller that supports the Dead Bus Recovery Mechanism shall also support:

- Dead Bus Recovery logic, in order to test the I3C Bus for an Active Controller that is both present and responsive:
  - A mechanism that uses the optional I3C Error Type DBR flow for recovering an I3C Bus that does not currently have a responsive Active Controller;
  - A limited START Request mechanism, which pulls SDA Low and uses a timer to test for SCL going Low in response; and
  - An optional IBI (In-Band Interrupt) Initiator with Address Arbitration mechanism, which drives a configurable I3C Address onto the Bus if and only if SCL goes Low (i.e., a completed START)
- Dead Bus Recovery FSM, in order to receive commands from the Driver that engage the Dead Bus Recovery logic and conditionally enable the Host Controller in Active Controller mode (i.e., as Active Controller) if no Active Controller is found to be both present and responsive;
- Registers that can be written by the Driver, to configure the Dead Bus Recovery logic and trigger the mechanism;

- Registers that can be read by the Driver, to read the status of the mechanism and determine whether the Host Controller was able to become Active Controller; and
- Alternate behavior to enable the Host Controller in Standby Controller mode (i.e., as a Secondary Controller, per **Section 6.17.1.3**) if another Active Controller is present and responsive
- Optional connection with vendor-defined capabilities for I3C Secondary Bus Controller Logic, or other I3C Target capabilities (if these are supported and enabled)
- Timer-based interrupt to provide the result of engaging the Dead Bus Recovery Mechanism, so the Host can determine whether the Host Controller became the Active Controller or joined the I3C Bus as a Secondary Controller (if supported)
- Extensions:
  - Specific Extended Capability structure (see **Section 7.7.6**) to expose the Dead Bus Recovery capabilities and features defined in this I3C HCI Feature Specification (i.e., for configuration)
  - Support for additional Internal Control Command sub-command (see **Section 8.4.2.8**) to engage the configured Dead Bus Recovery Mechanism by testing the I3C Bus, use the Dead Bus Recovery FSM and conditionally enter the appropriate mode based on the initial I3C role

**Figure 42** shows a high-level example of a Host Controller implementation that also supports the Dead Bus Recovery Mechanism. This figure is based on the basic Host Controller diagram in **Section 5.1** and adds DBR Logic, the FSM to manage the operational state of the I3C Bus Controller Logic, and additional HCI configuration registers that are specific to the Dead Bus Recovery Mechanism.



**Figure 42 Example Host Controller Implementation with Dead Bus Recovery Mechanism**

## 7 Register Interface

This Section of the Specification details the Register Set implemented by the Host Controller.

The Register Set has several sections. The Capabilities and Operational registers (*Section 7.3.1*) are mandatory to implement, and for data transfer either the PIO Section or the Ring Headers Section is mandatory to implement.

For Host Controllers that expose both PIO Mode and DMA Mode, the Driver shall select the current operating mode, and shall not use the register section associated with the other operating mode (the one that is not selected).

The DAT and DCT Sections shall either be implemented either in the Register Map, or accessed from Host system memory. If the Host Controller does not implement the DAT and/or the DCT as internal registers, then the Driver shall allocate memory and provide the address in the Base Address registers that hold pointers for Device Context (see *Section 7.4.19* and *Section 7.4.20*).

All Vendor-Specific registers should be exposed in vendor-specific Extended Capability structures, as part of the Extended Capabilities list (see *Section 7.7*).

The Host Controller shall ignore writes to Reserved fields and registers. The Host Controller Driver shall write the value 0 to all Reserved fields and registers. A Read from any Reserved field or register shall return the value 0.

The Host Controller shall support Byte, Word, and DWORD accesses to all registers in the Register Map.

### 7.1 Register Memory Access Conventions

In Memory Access columns in *Section 7*, the conventions shown in *Table 11* are used.

**Table 11 Register Memory Access Conventions**

Symbol	Meaning	Notes
R	Read-Only	Can only read from the register, cannot write to it
R/W	Read/Write	Can read from the register or write to it
R/cW	Read, conditional Write	Can read from the register Writes are conditionally allowed, based on operating mode or other current status
R/W1C	Read, Write 1 to Clear	Can read from the register To reset register value to its default value, write the value 1
W	Write-Only	Can only write to the register, read returns 0

### 7.2 PCI Configuration Registers

Per the *PCI Code and ID Assignment Specification [PCISIG01]*, compliance to this MIPI I3C Host Controller Interface Specification is indicated in the PCI header with Base Class equal to 0xCh, Sub Class equal to 0xAh, and Programming Interface equal to 0x0h. BAR0 configures decode for the HCI registers defined in the Register Map section. For more information refer to *Section 9.2.1.1*.

## 7.3 Register Map

The Host Controller implements a defined Register Map (i.e., a standardized set of registers) for configuration and control. When using System Bus types that support direct memory access (see [Section 9.1](#)), the Host Controller may also optionally support Driver-allocated Host memory; see the maximal configuration shown in [Figure 43](#).

The Register Set is divided into the following sections:

- **Capability and Operational Registers** (see [Section 7.3.1](#)) define software-discoverable Host Controller capabilities, allow software to limit these capabilities, control Host Controller behavior and current state, and define pointers to all other sections in the Register Map.

**Note:**

*Capability and Operational Registers are the only registers in the Register Map whose offset from Base Address is static. The offset for all other registers in the Register Map are instead related to the particular Register Map section. These section base offsets might be read from the Capability and Operational Registers section, or from the Extended Capabilities registers in any Extended Capability structures that might be defined. Software shall evaluate the section base offset before accessing any register outside of the Capability and Operational Registers section.*

- **Device Address Table (DAT)** (see [Section 7.3.3](#)) stores Static and Dynamic Addresses and configuration information for some or all Devices controlled by the Host Controller. The DAT table is populated by software.

**Note:**

*An index into the DAT is used for target Device identification in Command/Transfer Descriptors and the Host Controller shall use the index into the DAT to determine the Device's Address from the indicated DAT entry for all Transfer Commands.*

- **Device Characteristic Table (DCT)** (see [Section 7.3.3](#)) is a transient table that provides temporary storage and captures the PID, DCR, BCR, and assigned Dynamic Address during the Dynamic Address Allocation procedure (i.e., the [ENTDAA](#) CCC), for software to read afterwards.
- Optional **Device Context** in Driver-allocated Host system memory stores the DAT, the DCT, or both, if supported by the System Bus.
- **Ring Headers Descriptor** and a set of **Ring Headers** (see [Section 7.3.4](#)) define the header structure for Rings. Rings are used to hold descriptors for DMA transfers for Commands, Transfers, and Events. This Specification permits multiple Ring Header instances.
- **PIO (Programmable I/O)** (see [Section 7.3.5](#)) defines registers to allow Command, Transfer, and Event transmission using registers rather than Rings.
- **Extended Capabilities** (see [Section 7.3.2](#)) define a linked list of capabilities, including vendor-specific capabilities, arranged as a linked list of Extended Capability structures.

**Note:**

*Any features beyond those defined in this Specification would be implemented through vendor-specific Extended Capability structures.*

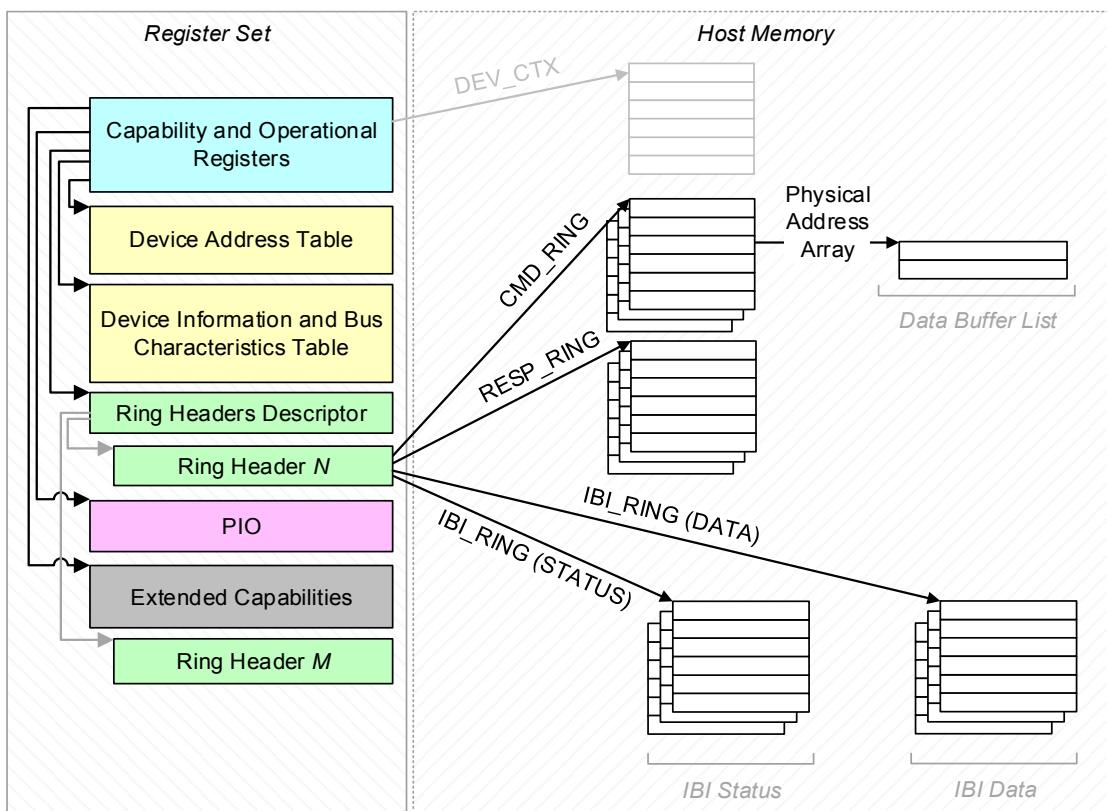


Figure 43 I3C Host Controller Interface Overview

5473 Although the Device Address Table (DAT) and Device Characteristics Table (DCT) are logically  
5474 considered structures, from the software point of view they should be implemented as a set of registers for  
5475 efficiency. If the system bus type supports direct memory access and the Host Controller includes DMA  
5476 capability, then the Host Controller shall indicate whether the DAT and/or DCT are implemented as  
5477 registers, or whether the Driver must allocate memory for all DAT entries and/or all DCT entries. This  
5478 memory is referred to as the Device Context.

5479 If the Host Controller has multiple Bus Controller instances (per *Section 7.7.4*), then some or all of these  
5480 register sections will exist for each Bus Controller instance.

### 7.3.1 Capabilities and Operation Registers

Registers in the Capabilities and Operation Registers section are defined with offsets from the Register Map base Address of the current Bus Controller instance. The base Address of the first Bus Controller instance shall always be 0x0. Base Addresses of any additional Bus Controller instances in the same Host Controller are exposed by the registers in the Multi-Bus Instance Extended Capability Section (see [Section 7.7.4](#)).

All other register sections that are associated with this I3C Bus Controller instance in the Host Controller may be placed at any offset relative to this base Address, as indicated by the pointers (i.e., section offset registers) within this register section (see [Section 7.4](#)):

- Register **DAT\_SECTION\_OFFSET** (see [Section 7.4.11](#)) shall describe the DAT, whether it is implemented as registers in the Register Set, or required to be allocated by the Driver as Host system memory (as per [Section 7.3.3](#) and [Section 8.1](#)).
- Register **DCT\_SECTION\_OFFSET** (see [Section 7.4.12](#)) shall describe the DCT, whether it is implemented as registers in the Register Set, or required to be allocated by the Driver as Host system memory (as per [Section 7.3.3](#) and [Section 8.2](#)).
- If this I3C Bus Controller instance supports DMA Mode, then register **RING\_HEADERS\_SECTION\_OFFSET** (see [Section 7.4.13](#)) shall provide the offset to the Ring Header Preamble at the start of the Ring Header Specific Registers section (see [Section 7.3.4](#)).
- If this I3C Bus Controller instance supports PIO Mode, then register **PIO\_SECTION\_OFFSET** (see [Section 7.4.14](#)) shall provide the offset to the start of the Programmable I/O Registers section (see [Section 7.3.5](#)).
- Register **EXT\_CAPS\_SECTION\_OFFSET** (see [Section 7.4.15](#)) shall provide the offset to the start of the first Extended Capability structure (i.e., the beginning of the linked list of such structures; see [Section 7.3.2](#)).

The definition of the Capabilities and Operation Registers section is specific to the HCI version indicated by register **HCI\_VERSION** (i.e., the first DWORD register). Naturally, the Host Controller may define additional, vendor-specific registers in addition to the mandatory definitions.

This version of the I3C HCI Specification defines the Capabilities and Operation Registers used in Active Controller mode, and that are generally used while the Host Controller is acting as Active Controller of the I3C Bus. Support for Target and Secondary Controller use models (i.e., for Standby Controller mode) is optional, and would require additional registers; this I3C HCI Specification defines certain Extended Capability structures for Standby Controller mode (see [Section 6.17](#) and [Section 7.7.11](#)).

**Table 12** summarizes the Capabilities and Operation section of the Register Map.

5512

**Table 12 Register Map: Capabilities and Operation Registers**

Offset	Offset + 0x0	Offset + 0x4	Offset + 0x8	Offset + 0xC
0x00	HCI_VERSION	HC_CONTROL	CONTROLLER_DEVICE_ADDR	HC_CAPABILITIES
0x10	RESET_CONTROL	PRESENT_STATE	-	-
0x20	INTR_STATUS	INTR_STATUS_ENABLE	INTR_SIGNAL_ENABLE	INTR_FORCE
0x30	DAT_SECTION_OFFSET	DCT_SECTION_OFFSET	RING_HEADERS_SECTION_OFFSET	PIO_SECTION_OFFSET
0x40	EXT_CAPS SECTION_OFFSET	-	-	INT_CTRL_CMDS_EN
0x50	-	-	IBI_NOTIFY_CTRL	IBI_DATA_ABORT_CTRL [New in v1.2]
0x60	DEV_CTX_BASE_LO	DEV_CTX_BASE_HI	DEV_CTX_SG	-
0x70	-	-	-	-

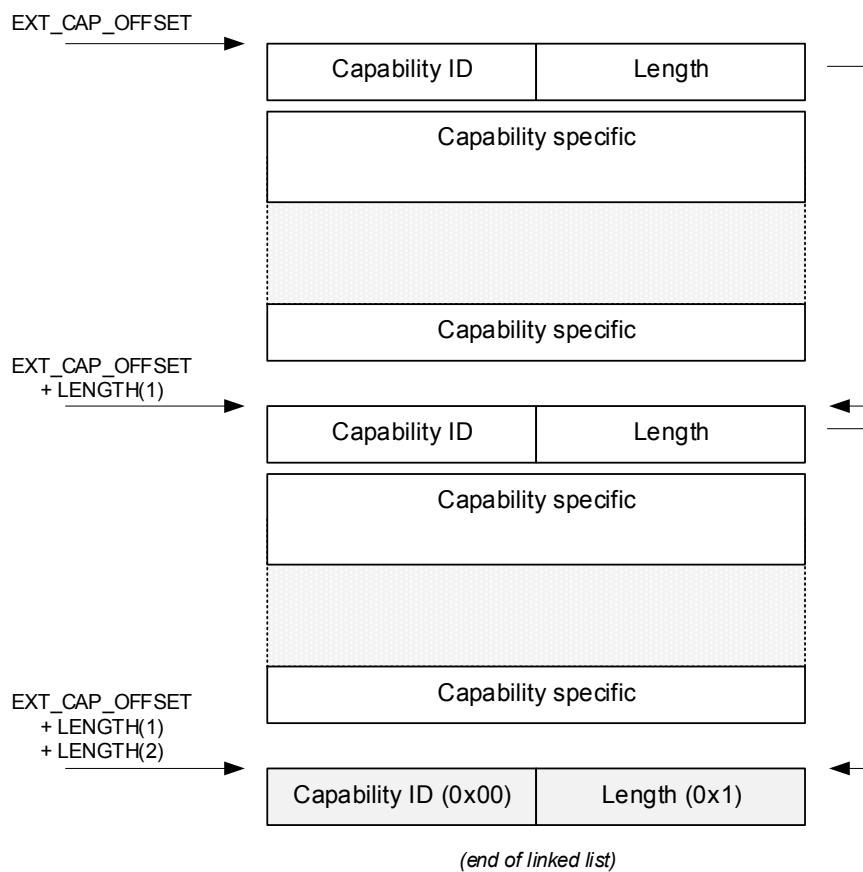
### 7.3.2 Extended Capabilities Registers

Extended Capabilities are expressed as a list of vendor-specific registers. These registers shall be placed in a contiguous set of registers, and shall be treated as a linked list comprising one or more Extended Capability structures.

In general, the Extended Capability structures that are defined in this I3C HCI Specification are optional for an I3C Host Controller; however, some exceptions are defined in order to expose support for certain optional features and capabilities.

Additionally, certain optional features and capabilities might be subsequently defined in future I3C HCI Feature Specifications, which would extend the core Host Controller functionality defined in this I3C HCI Specification.

Each Extended Capability structure has a defined length in DWORDS, and specifies a Capability ID defining the type of capability the structure describes. Note that the first register in each Extended Capability structure must be an Extended Capability Header register ([EXTCAP\\_HEADER](#), *Section 7.7.1*) since it defines the structure of the linked list.



**Figure 44 Concept of Linked List of Extended Capability Structures**

5527 In **Table 13**, the Extended Capability Registers are shown as a flat array of registers, with the Offset starting  
5528 at some relative location in the register map for the Host Controller. The actual starting Offset of the first  
5529 register shall be relative to the value indicated in the table, added to the value in field **SECTION\_OFFSET** of  
5530 register **EXT\_CAPS\_SECTION\_OFFSET** (*Section 7.4.15*).

5531 **Table 13 Extended Capability Mandatory Registers**

Offset	Offset + 0x0	Offset + 0x4	Offset + 0x8	Offset + 0xC
+0x00	<b>EXTCAP_HEADER</b>	-	-	-
+0x10	-	-	-	-
+0x20	-	-	-	-
+0x30	-	-	-	-

5532 Since the Extended Capability structures are arranged as a linked list, the end of the list must be indicated  
5533 by an end-of-list marker. This marker also takes the form of a special Extended Capability Header register  
5534 (*Section 7.7.1*) following the last valid Extended Capability structure, which has a Capability ID of 0x00,  
5535 and a defined length of 1 DWORD (i.e., the structure only includes this one register).

5536 This specification defines a number of supported Capability IDs that may be used for each structure. The  
5537 intended use for each Capability ID is described within the subsections of *Section 7.7*, along with the  
5538 defined format for the associated type of Extended Capabilities section for that Capability ID.

5539 In most cases, a structure for a particular Capability ID may be instantiated only once within the linked list  
5540 of Extended Capability structures; however, some exceptions are defined for specific Capability IDs which  
5541 may have one or more instances within the linked list of structures.

### 7.3.3 DAT and DCT Registers

5542 The Device Address Table (DAT) and Device Characteristic Table (DCT) shall be implemented either as a  
5543 Register Set, or in Driver-allocated memory at an Address indicated with the Device Context pointer.

5544 When the DAT and/or DCT is implemented in a Register Set, the order of DWORDs in the structure is  
5545 DW0 (comprising Bits[31:0]) first (lowest Address). All entries within the register set are contiguous.

5546 **Example:** For 2 DWORDs DAT entry size, if the DAT offset points at 0x400 for the first Device,  
5547 then the lowest-order DWORD of the DAT (DW0, containing Device type, Static Address, etc.) is  
5548 located at offset 0x400, and the next DWORD of the DAT (DW1, containing the Auto-Command  
5549 configuration) is located at offset 0x404. The second Device is located at offsets 0x408 (DW0) and  
5550 0x40C (DW1).

5551 When the DAT and/or DCT is implemented in Driver-allocated memory (per *Section 6.2*), the DWORDs  
5552 appear with DW0 in the lowest Addresses. The Host Controller shall define the expected sizes for the tables  
5553 based on the number of entries (i.e., field **TABLE\_SIZE**). The Host Controller shall use the following  
5554 partitioning for the Driver-allocated memory:

- 5555 • If **only** the DAT is implemented in Driver-allocated memory (i.e., not registers), then:
  - 5556 • The first DAT entry shall start at offset 0x0; the second DAT entry shall start at offset 0x8; etc. This  
5557 continues for all entries, as defined by field **TABLE\_SIZE** in register **DAT\_SECTION\_OFFSET** (i.e., with 2  
5558 DWORDs for each entry).
  - 5559 • The Driver must allocate a region of memory that is large enough to hold all DAT entries.
- 5560 • If **only** the DCT is implemented in Driver-allocated memory (i.e., not registers), then:

- 5561     • The first DCT entry shall start at offset 0x0; the second DCT entry shall start at offset 0x10; etc. This  
5562       continues for all entries, as defined by field **TABLE\_SIZE** in register **DCT\_SECTION\_OFFSET** (i.e., with 4  
5563       DWORDs for each entry).  
5564     • The Driver must allocate a region of memory that is large enough to hold all DCT entries.  
5565     • If **both** the DAT and DCT are implemented in Driver-allocated memory (i.e., not registers), then:  
5566       • The DAT entries shall start at offset 0x0 (as above), with 2 DWORDs for each DAT entry, and continue  
5567       through the last DAT entry (i.e., as defined by field **TABLE\_SIZE** in register **DAT\_SECTION\_OFFSET**).  
5568       • After the last DAT entry, the DCT entries shall start at offset (**DAT\_SECTION\_OFFSET**.**TABLE\_SIZE**) \* 8,  
5569       with 4 DWORDs for each DCT entry, and continue through the last DCT entry (i.e., as defined by field  
5570       **TABLE\_SIZE** in register **DCT\_SECTION\_OFFSET**).  
5571       • The Driver must allocate a region of memory that is large enough to hold all DAT and DCT entries.

5572     For detailed descriptions of the DAT and DCT data structures, see *Section 8.1* and *Section 8.2*, respectively.

5573     **Note:**

5574       *The **TABLE\_SIZE** fields in registers **DAT\_SECTION\_OFFSET** and **DCT\_SECTION\_OFFSET** are read-*  
5575       *only to the Host, and cannot be changed by the Driver. The Host Controller determines how many*  
5576       *entries it uses for each such table that it expects the Driver to store in system memory.*

### 7.3.4 Ring Headers Registers

If DMA is implemented, then the Ring Headers registers section shall be implemented. The Ring Header Preamble registers shall contain the offsets for each Ring Header (i.e., one for each supported Ring Bundle).

**Table 14 Ring Header Preamble**

Offset	Offset + 0x0	Offset + 0x4	Offset + 0x8	Offset + 0xC
0x00	RHS_CONTROL	RH0_OFFSET	RH1_OFFSET	RH2_OFFSET
0x10	RH3_OFFSET	RH4_OFFSET	RH5_OFFSET	RH6_OFFSET
0x20	RH7_OFFSET	-	-	-
0x30	-	-	-	-

**Table 15 Ring Header (Per Ring Bundle)**

Offset	Offset + 0x0	Offset + 0x4	Offset + 0x8	Offset + 0xC
0x00	CR_SETUP	IBI_SETUP	CHUNK_CONTROL	-
0x10	RH_INTR_STATUS	RH_INTR_STATUS_ENABLE	RH_INTR_SIGNAL_ENABLE	RH_INTR_FORCE
0x20	RH_STATUS	RH_CONTROL	RH_OPERATION1	RH_OPERATION2
0x30	RH_CMD_RING_BASE_LO	RH_CMD_RING_BASE_HI	RH_RESP_RING_BASE_LO	RH_RESP_RING_BASE_HI
0x40	RH_IBI_STATUS_RING_BASE_LO	RH_IBI_STATUS_RING_BASE_HI	RH_IBI_DATA_RING_BASE_LO	RH_IBI_DATA_RING_BASE_HI
0x50	RH_CMD_RING_SG	RH_RESP_RING_SG	RH_IBI_STATUS_RING_SG	RH_IBI_DATA_RING_SG
0x60	-	-	-	-

### 7.3.5 Programmable I/O Registers

5582 PIO Mode can be used for simplified implementations that implement register-based processing of  
 5583 commands, transfers, and events. For PIO Mode, single queues are defined for Commands, Transfers,  
 5584 Responses, and Interrupts.

5585 If PIO Mode is supported, then the PIO registers section shall be implemented.

5586

**Table 16 Register Map: PIO Access Area**

Offset	Offset + 0x0	Offset + 0x4	Offset + 0x8	Offset + 0xC
0x00	COMMAND_QUEUE_PORT	RESPONSE_QUEUE_PORT	XFER_DATA_PORT	IBI_PORT
0x10	QUEUE_THLD_CTRL	DATA_BUFFER_THLD_CTRL	QUEUE_SIZE	ALT_QUEUE_SIZE [New in v1.2]
0x20	PIO_INTR_STATUS	PIO_INTR_STATUS_ENABLE	PIO_INTR_SIGNAL_ENABLE	PIO_INTR_FORCE
0x30	PIO_CONTROL [New in v1.2]	-	-	-

## 7.4 Host Controller Capability & Operations Registers

The Host Controller Capability & Operations Registers section is defined at offsets 0x0 through 0x7C from the BASE Address of the Bus Controller instance.

### 7.4.1 HCI Version (HCI\_VERSION) (BASE + 0x0)

This register indicates the version number of the I3C HCI Specification (this Specification) that the Host Controller implements. The definition of this register shall not change across all versions of the I3C HCI Specification.

Table 17 HCI Version (HCI\_VERSION) Register

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	VERSION	R	0x0	<b>HCI Version</b> Major/Minor version of I3C HCI that the Host Controller implements. Lowest 8 bits encode the minor number and the revision. Each 4 bits in this field are encoded as binary encoded decimals. <b>Examples:</b> <ul style="list-style-type: none"><li>• 0x00000100: HCI v1.0</li><li>• 0x00000110: HCI v1.1</li><li>• 0x00000120: HCI v1.2 (this Specification version)</li><li>• 0x00000200: HCI v2.0</li></ul> Several revisions may be skipped.

#### 7.4.2 Host Controller Control (HC\_CONTROL) (BASE + 0x4)

5593 Host Controller Control register is used to manage the Host Controller and Controller Configuration.

5594 **Table 18 Host Controller Control (HC\_CONTROL) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [31]	<b>BUS_ENABLE</b>	R/W	0x0	<p><b>Host Controller Bus Enable</b>      Enables or disables the operation on the I3C Bus by the Host Controller.      If the software sets this bit, then it also confirms that initialization is done, and that the Host Controller can use the programmed register values (e.g., generation of SCL on IBI detection, etc.). If this bit is not set, then the Host Controller shall not generate SCL for incoming IBI.      Software may disable the Host Controller bus operation while it is active, However:</p> <ul style="list-style-type: none"> <li>• If a disable request occurs while receiving IBI, the actual disabling will not occur until reception of the IBI is complete.</li> <li>• If a disable request occurs while the Host Controller still has additional Commands queued for transfers, then the actual disabling will not occur until transmission of all queued Commands is complete (i.e., until the Host Controller encounters a Command with field <b>TOC</b> set to 1). The Host Controller shall then return to the Idle state (as indicated in register <b>PRESENT_STATE_DEBUG</b>).</li> </ul> <p>When the software reads the value 1'b0 from this field, this indicates that the Host Controller bus operation disable operation has completed.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>1'b0: DISABLED:</b> The Host Controller bus operation is Disabled</li> <li>• <b>1'b1: ENABLED:</b> The Host Controller bus operation is Enabled</li> </ul>

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [30]	RESUME	R/W1C	0x0	<p><b>Host Controller Resume</b>  This bit is used to resume Host Controller operation following the Halt state.  The Host Controller enters the Halt state (as indicated in register <b>PRESENT_STATE_DEBUG</b>) as a result of any type of error occurring in a transfer. The error type is indicated by field <b>ERR_STATUS</b> in the Response Descriptor (see <b>Section 8.5</b>).  After the Host Controller has entered the Halt state, software must write the value 1'b1 to the <b>RESUME</b> bit to resume operation (i.e., write 1 to clear).  <b>Values</b> (when read): <ul style="list-style-type: none"> <li>• 1'b0: <b>RUNNING</b>: The Host Controller is running</li> <li>• 1'b1: <b>SUSPENDED</b>: The Host Controller is suspended</li> </ul> </p>
1 [29]	ABORT	R/W	0x0	<p><b>Host Controller Abort</b>  When set to 1, this bit triggers the Host Controller Abort operation which may abort the currently issued transfer (see <b>Section 6.8.4</b>).  In response to an <b>ABORT</b> request, the Host Controller terminates the current transaction on the I3C Bus at the nearest opportunity (i.e., after the complete data byte is transferred or received).  Usage and operational details vary, per the current operating mode. (See <b>Section 6.8.4, Table 6</b>)</p>
16 [28:13]	RESERVED	R	0x0	–
1 [12]	HALT_ON_CMD_SEQ_TIMEOUT	R/W	0x0	<p><b>Halt on Command Sequence Timeout</b>  This bit acts as global control to halt Host Controller operations if a command sequence timeout condition occurs, and this causes such a sequence to be terminated (see <b>Section 6.13.2</b>).  <b>Values:</b> <ul style="list-style-type: none"> <li>• 1'b0: <b>DISABLE</b>: Continue processing after termination</li> <li>• 1'b1: <b>ENABLE</b>: Halt processing, enter the Halt state indicated with field <b>RESUME</b></li> </ul> </p>
3 [11:9]	RESERVED	R	0x0	–

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [8]	HOT_JOIN_CTRL	R/W	0x0	<p><b>Hot-Join ACK/NACK Control</b></p> <p>This bit acts as global control to either ACK (1'b0) or NACK (1'b1) all Hot-Join Requests arriving from the Devices on the I3C Bus. If set to NACK (1'b1), then the NACK will be followed by the Broadcast CCC to disable Hot-Join.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: ACK: ACK the Hot-Join request</li> <li>• 1'b1: NACK: NACK and send Broadcast CCC to disable Hot-Join</li> </ul>
1 [7]	I2C_DEV_PRESENT	R/W	0x0	<p><b>I<sup>2</sup>C Device Present on Bus</b></p> <p>Indicates whether or not any Legacy I<sup>2</sup>C Devices are present on the I3C Bus. Whenever this bit is set, the Host Controller shall use the HDR-TSL protocol (Ternary Symbol Legacy Inclusive Bus) for HDR-TS Transfers.</p> <p>The specific Mode of the transfer (SDR, DDR, HDR-TS) is controlled on a per-Command basis. The Host Controller shall use this field to select either TSL (1'b1) or TSP (1'b0) for HDR-TS transfers.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: DISABLED: Legacy inclusive Bus Mode disabled</li> <li>• 1'b1: ENABLED: Legacy inclusive (mix) Bus Mode enabled</li> </ul>

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [6]	MODE_SELECTOR	R/cW (or R)	IMPL	<p><b>DMA/PIO Mode Selector</b>            Indicates the current operating mode (DMA or PIO) of the Host Controller for its Controller Logic (i.e., Active Controller mode).            This configuration shall only be changed when the Host Controller is in DISABLED state. The Driver shall explicitly set this bit to indicate the desired operating mode.</p> <ul style="list-style-type: none"> <li>• If only one operating mode is supported, then this bit shall be Read-Only.</li> <li>• If both operating modes are supported, then PIO Mode shall be the default (i.e., 1'b1 reset value).</li> </ul> <p>This register is writeable, only if both PIO Mode and DMA Mode are supported, and the Host Controller implements both Register Sets.</p> <ul style="list-style-type: none"> <li>• PIO Mode can only be selected if register <b>PIO_SECTION_OFFSET</b> is valid.</li> <li>• DMA Mode can only be selected if register <b>RING_HEADERS_SECTION_OFFSET</b> is valid.</li> </ul> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: DMA: DMA Mode enabled</li> <li>• 1'b1: PIO: PIO Mode enabled</li> </ul>
1 [5]	RESERVED	R	0x0	—
1 [4]	DATA_BYTE_ORDER_MODE	R	IMPL	<p><b>Data Byte Ordering Mode</b>            This field indicates the supported byte ordering mode, per <b>Section 6.8.3</b>.            An implementer may choose the byte ordering mode according to the expected byte ordering of the Host system.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: LittleEndian ordering (i.e., the byte at Bits[7:0] is first)</li> <li>• 1'b1: BigEndian ordering (i.e., the byte at Bits[31:24] is first)</li> </ul>

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [3]	<b>AUTOCMD_DATA_RPT</b> <i>(New for I3C HCI v1.2 – see Note after this Table)</i>	R/W	IMPL	<p><b>Auto-Command Data Report</b> This bit controls whether the Host Controller reports Read transfer data from an Auto-Command in the same report as the preceding IBI (i.e., a Pending Read Notification) or whether these are reported separately.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: Coalesced reporting, include Auto-Command data after last IBI payload data byte</li> <li>• 1'b1: Separated reporting, use new IBI Status Descriptor for Auto-Command data with field <b>STATUS_TYPE</b> set to 3'b100</li> </ul> <p><b>Note:</b> <i>Value 1'b0 is the only option defined by I3C HCI version 1.1 and earlier, which always coalesced the Auto-Command data after the last IBI payload data byte.</i></p>
2 [2:1]	RESERVED	R	0x0	–
1 [0]	<b>IBA_INCLUDE</b>	R/W	0'b0	<p><b>Include I3C Broadcast Address</b> This bit controls whether the I3C Broadcast Address (7'h7E) is included for private transfers. If the I3C Broadcast Address is not included for private transfers, then IBIs driven from Target Devices might not win the Arbitration, potentially delaying acceptance of the IBIs.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: Do not include I3C Broadcast Address for Private Transfers</li> <li>• 1'b1: Include I3C Broadcast Address for Private Transfers</li> </ul>

#### 7.4.3 Controller Device Address (CONTROLLER\_DEVICE\_ADDR) (BASE + 0x8)

The Controller Device Address register is used to program the Host Controller's Dynamic Address. In Active Controller mode, the software shall self-assign the Dynamic Address if it is the Primary Controller (i.e., before initiating its first Controller-to-Controller Handoff Procedure).

**Table 19 Controller Device Address (CONTROLLER\_DEVICE\_ADDR) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [31]	DYNAMIC_ADDR_VALID	R/W	0x0	<p><b>Dynamic Address is Valid</b></p> <p>This bit indicates whether or not the value in the DYNAMIC_ADDR field is valid.</p> <p>This bit must be set to 1'b1 before passing the Controller Role to another Controller.</p> <p>In Active Controller mode, the software shall set this bit to 1'b1 as it self-assigns its Dynamic Address. This requires the Host Controller to act as the Primary Controller of the I3C Bus.</p> <p>If operating as a Secondary Controller (i.e., in Standby Controller mode), then the Host Controller shall set this bit to 1'b1 when the Active Controller assigns the Dynamic Address (see Note below this table).</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: The Dynamic Address field is not valid</li> <li>• 1'b1: The Dynamic Address field is valid</li> </ul>
8 [30:23]	RESERVED	R	0x0	–
7 [22:16]	DYNAMIC_ADDR	R/W	0x0	<p><b>Device Dynamic Address</b></p> <p>This field is used to program the Host Controller Device's Dynamic Address in Active Controller mode, before passing the Controller Role to another Controller.</p> <p>In Active Controller mode, the software shall program the Dynamic Address as it self-assigns its Dynamic Address.</p> <p>If operating as a Secondary Controller, then the Primary Controller shall assign the Dynamic Address (i.e., during the <a href="#">ENTDAA</a> or <a href="#">SETDASA/SETAASA</a> procedure) or when an Active Controller changes the Dynamic Address (i.e., using the <a href="#">SETNEWDA</a> CCC).</p> <p>This field contains a valid Dynamic Address when field DYNAMIC_ADDR_VALID contains the value 1'b1.</p>
16 [15:0]	RESERVED	R	0x0	–

**Note:**

*This register shall only be written when the Host Controller is operating in Active Controller mode. If the Host Controller is operating as a Secondary Controller (i.e., in Standby Controller mode), then*

5602      *this register shall be read-only, and the contents shall be set by other logic within the Host  
5603      Controller.*

#### 7.4.4 Host Controller Capabilities (HC\_CAPABILITIES) (BASE + 0xC)

The Hardware Controller Capabilities register identifies the capabilities of the Controller Host Controller hardware.

**Table 20 Host Controller Capabilities (HC\_CAPABILITIES) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [31]	RESERVED	R	0x0	—
1 [30]	SG_CAPABILITY_DC_EN	R	IMPL	Defines whether the Host Controller supports Scatter-Gather List for the indicated capability, as described in <b>Section 6.2.1</b> . <b>Values:</b> <ul style="list-style-type: none"><li>• 1'b0: DISABLED: Scatter-Gather is not supported for this capability</li><li>• 1'b1: ENABLED: Scatter-Gather is supported for this capability</li></ul>
1 [29]	SG_CAPABILITY_IBI_EN	R	IMPL	—
1 [28]	SG_CAPABILITY_CR_EN	R	IMPL	—
6 [27:22]	RESERVED	R	0x0	—
2 [21:20]	CMD_SIZE	R	IMPL	Defines the size and structure of the Command Descriptor supported by the Host Controller. <b>Values:</b> <ul style="list-style-type: none"><li>• 2'b00: 2 DWORDs</li><li>• All other values: Reserved for future use</li></ul> See <b>Section 6.7</b> and <b>Section 8.4</b> .
7 [19:14]	RESERVED	R	0x0	—
1 [13]	SCHEDULED_COMMANDS_EN <i>(New for I3C HCI v1.2)</i>	R	IMPL	Defines whether the Host Controller supports Scheduled Commands capabilities per <b>Section 6.16</b> . This capability bit shall only be applicable for I3C HCI versions ≥ v1.2. <b>Values:</b> <ul style="list-style-type: none"><li>• 1'b0: DISABLED: Scheduled Commands not supported</li><li>• 1'b1: ENABLED: Scheduled Command are supported, with at least one Extended Capability structure</li></ul>

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [12]	<b>IBI_CREDIT_COUNT_EN</b> <i>(New for I3C HCI v1.2)</i>	R	IMPL	Defines whether the Host Controller supports Target IBI Credit Counting per <b>Section 6.9.5</b> . If this capability bit is set to 1'b1, then the Driver may enable credit counting and manage credit via the Extended Capabilities structure (see <b>Section 7.7.5</b> ). This capability bit shall only be applicable for I3C HCI versions ≥ v1.2.
1 [11]	<b>IBI_DATA_ABORT_EN</b> <i>(New for I3C HCI v1.2)</i>	R	IMPL	Defines whether the Host Controller supports the IBI Data Abort operation for ongoing IBI data payloads per <b>Section 6.9.4</b> . If this capability bit is set to 1'b1, then the Driver may control the monitoring logic by writing to register <b>IBI_DATA_ABORT_CTRL</b> (see <b>Section 7.4.18</b> ). This capability bit shall only be applicable for I3C HCI versions ≥ v1.2.
1 [10]	<b>CMD_CCC_DEFBYTE</b>	R	0x1	Defines whether the Host Controller supports Transfer Commands that indicate CCCs with Defining Bytes, using the managed CCC transfer framing model defined in <b>Section 6.3</b> of the I3C TCRI Specification [ <b>MIPI06</b> ]. This capability bit shall only be applicable for I3C HCI versions ≥ v1.1. A value of 1'b1 indicates support, which is required for HCI version v1.1 and greater.
2 [9:8]	RESERVED	R	0x0	—
1 [7]	<b>HDR_TS_EN</b>	R	IMPL	Defines whether the Host Controller supports HDR-Ternary transfers. Selection of HDR-TSP vs. HDR-TSL depends on the value of field <b>I2C_DEV_PRESENT</b> in register <b>HC_CONTROL</b> (see <b>Section 7.4.2</b> ). Values: <ul style="list-style-type: none"> <li>• 1'b0: <b>DISABLED</b>: HDR-Ternary is not supported</li> <li>• 1'b1: <b>ENABLED</b>: HDR-Ternary is supported</li> </ul>

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [6]	HDR_DDR_EN	R	IMPL	<p>Defines whether the Host Controller supports HDR-DDR transfers.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: <b>DISABLED</b>: HDR-DDR is not supported</li> <li>• 1'b1: <b>ENABLED</b>: HDR-DDR is supported</li> </ul>
1 [5]	STANDBY_CR_CAP	R	IMPL	<p>Defines whether the Host Controller supports handoff of the Active Controller role to another Device on the I3C Bus, including Standby Controller mode per <b>Section 6.17</b>.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: <b>DISABLED</b>: Not capable of acting as Standby Controller</li> <li>• 1'b1: <b>ENABLED</b>: Capable of acting as Standby Controller (i.e., Secondary Controller)</li> </ul> <p><b>Note:</b>  <i>Secondary Controller capabilities (i.e., value 1'b1) would require implementer to support the Extended Capability structure (Section 7.7.11).</i></p>
1 [4]	RESERVED	R	0x0	—
1 [3]	AUTO_COMMAND	R	IMPL	<p>Defines whether the Host Controller supports Auto-Command functionality per <b>Section 6.11</b>.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: <b>DISABLED</b>: Auto-Command is not supported</li> <li>• 1'b1: <b>ENABLED</b>: Auto-Command is supported</li> </ul>
1 [2]	COMBO_COMMAND	R	IMPL	<p>Defines whether the Host Controller supports Combo Transfer Command transfers.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: <b>DISABLED</b>: Combo Transfer is not supported</li> <li>• 1'b1: <b>ENABLED</b>: Combo Transfer is supported</li> </ul>
2 [1:0]	RESERVED	R	0x0	—

#### 7.4.5 Reset Control (RESET\_CONTROL) (BASE + 0x10)

5607 Reset Control register is used to reset specific functional areas of Host Controller, including buffer resets.

5608 **Table 21 Reset Control (RESET\_CONTROL) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
<b>26</b> [31:6]	RESERVED	-	-	-
<b>1</b> [5]	IBI_QUEUE_RST	R/W	0x0	<b>IBI Queue Buffer Software Reset</b> On software setting this bit to 1'b1, the IBI Queues in the Controller shall be flushed. This field shall be cleared automatically upon completion of IBI Queue reset. This field is valid for PIO Mode only.
<b>1</b> [4]	RX_FIFO_RST	R/W	0x0	<b>Receive Queue Buffer Software Reset</b> On software setting this bit to 1'b1, the Rx Queues in the Controller shall be flushed. This field shall be cleared automatically upon completion of Rx Data Queue reset. This field is valid for PIO Mode only.
<b>1</b> [3]	TX_FIFO_RST	R/W	0x0	<b>Transmit Queue Buffer Software Reset</b> On software setting this bit to 1'b1, the Tx Queues in the Controller shall be flushed. This field shall be cleared automatically upon Tx Data Queue reset completion. This field is valid for PIO Mode only.
<b>1</b> [2]	RESP_QUEUE_RST	R/W	0x0	<b>Response Queue Software Reset</b> On software setting this bit to 1'b1, the Response Queues in the Controller shall be flushed. This field shall be cleared automatically upon Response Queue reset completion. This field is valid for PIO Mode only.
<b>1</b> [1]	CMD_QUEUE_RST	R/W	0x0	<b>Command Queue Software Reset</b> On software setting this bit to 1'b1, the Command Queues in the Controller shall be flushed. This field shall be cleared automatically upon Command Queue reset completion. This field is valid for PIO Mode only.

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [0]	SOFT_RST	R/W	0x0	<p><b>Core Software Reset</b></p> <p>On Driver setting this bit to 1'b1, the Host Controller shall be reset and disabled. All registers shall return to their reset values, and the software shall re-initialize the Controller. This field is cleared automatically upon Host Controller reset completion.</p> <p>This field also resets all Queues in the Host Controller.</p> <p>This field does not reset any Ring Headers in the Host Controller.</p> <p><b>Note:</b></p> <p><i>Programming this field while it contains a value of 1'b1 may result in undefined behavior.</i></p>

#### 7.4.6 Present State (PRESENT\_STATE) (BASE + 0x14)

5609  
5610 The Present State Register returns the Host Controller's current state. State has two parts: this register  
5611 which is mandatory, and an additional optional **PRESENT\_STATE\_DEBUG** register intended for debug  
purposes (see the Debug Capability registers in the Extended Capabilities list).

5612

**Table 22 Present State (PRESENT\_STATE) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
<b>29</b> [31:3]	RESERVED	R	0x0	–
<b>1</b> [2]	<b>AC_CURRENT_OWN</b>	R	0x0	<p><b>Active Controller</b>  This bit indicates whether or not the Host Controller is presently the Active Controller on the I3C Bus.  The Active Controller is the Controller-capable Device on the I3C Bus that owns the SCL line.  If Standby Controller mode is supported, then this field is mirrored in register <b>STBY_CR_STATUS</b> (<b>Section 7.7.11.4</b>).  <b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>1'b0: NOT_BUS_OWNER:</b> The Host Controller is not the Active Controller, and must request and acquire the Controller Role before initiating any transfer.</li> <li>• <b>1'b1: BUS_OWNER:</b> The Host Controller is the Active Controller, and as a result can initiate transfers.</li> </ul> <p><b>Note:</b>  <i>Acting as a Secondary Controller (i.e., value 1'b0) requires the implementer to add I3C Secondary Controller Logic, and also add support for Standby Controller mode as an Extended Capability.</i></p>
<b>2</b> [1:0]	RESERVED	R	0x0	–

#### 7.4.7 Interrupt Status (INTR\_STATUS) (BASE + 0x20)

The Interrupt Status register reflects the status of outstanding interrupts relating to general Host Controller errors or other actions.

The status fields are R/W1C (i.e., write 1 to clear).

The bit fields in this register act as Status Bits, according to the interrupt register model described in [Section 6.14](#). These bits will be set (i.e., the “Status bit set” action will occur) if the interrupt condition is detected, and if the corresponding bit in the Interrupt Status Enable Mask is set to 1'b1.

**Table 23 Interrupt Status (INTR\_STATUS) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
17 [31:15]	RESERVED	—	—	—
1 [14]	SCHED_CMD_MISSED_TICK_STAT	R/W1C	0x0	<b>Scheduled Command Missed Tick</b> The Scheduled Commands logic missed a tick interval because the I3C Bus Controller Logic was busy processing other scheduled Transfer Commands. See <a href="#">Section 6.16.1</a> .
1 [13]	HC_ERR_CMD_SEQ_TIMEOUT_STAT	R/W1C	0x0	<b>Host Controller Command Sequence Stall/Timeout</b> The I3C Bus Controller Logic experienced a command sequence stall condition (warning) or timeout (error). See <a href="#">Section 6.13.2</a> .
1 [12]	HC_WARN_CMD_SEQ_STALL_STAT	R/W1C	0x0	—
1 [11]	HC_SEQ_CANCEL_STAT	R/W1C	0x0	<b>Host Controller Cancelled Transaction Sequence</b> The I3C Bus Controller Logic was forced to cancel a transaction sequence.
1 [10]	HC_INTERNAL_ERR_STAT	R/W1C	0x0	<b>Host Controller Internal Error</b> Reflects the interrupt status that is populated based on any non-recoverable internal error of the Host Controller
10 [9:0]	RESERVED	—	—	—

#### 7.4.8 Interrupt Status Enable (INTR\_STATUS\_ENABLE) (BASE + 0x24)

The Interrupt Status Enable register enables or disables reporting of outstanding interrupts.

The bit fields in this register act as Interrupt Status Enable Mask bits, according to the interrupt register model described in *Section 6.14*. Writes to these fields shall enable or disable the visibility of an interrupt condition in the Status Register. The value of each field shall determine whether the interrupt condition will be passed to the Interrupt Signal Mask to result in an Interrupt trigger; and will also determine whether software sees the interrupt condition using polling (i.e., without using an interrupt service routine).

**Table 24 Interrupt Status Enable (INTR\_STATUS\_ENABLE) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
17 [31:15]	RESERVED	—	—	—
1 [14]	SCHED_CMD_MISSED_TICK_STAT_EN	R/W	0x0	<b>Scheduled Command Missed Tick Status Enable</b> Enables the corresponding interrupt bit. When set to 1'b1, interrupts for the corresponding interrupt bits in register <b>INTR_STATUS</b> shall be logged.
1 [13]	HC_ERR_CMD_SEQ_TIMEOUT_STAT_EN	R/W	0x0	<b>Host Controller Command Sequence Stall/Timeout Status Enable</b>
1 [12]	HC_WARN_CMD_SEQ_STALL_STAT_EN	R/W	0x0	Enables the corresponding interrupt bit. When set to 1'b1, interrupts for the corresponding interrupt bits in register <b>INTR_STATUS</b> shall be logged.
1 [11]	HC_SEQ_CANCEL_STAT_EN	R/W	0x0	<b>Host Controller Cancelled Transaction Sequence Status Enable</b> Enables the corresponding interrupt bit. When set to 1'b1, <b>HC_SEQ_CANCEL_STAT_EN</b> interrupts shall be logged.
1 [10]	HC_INTERNAL_ERR_STAT_EN	R/W	0x0	<b>Host Controller Internal Error Status Enable</b> Enables the corresponding interrupt bit. When set to 1'b1, <b>HC_INTERNAL_ERR_STAT</b> interrupts shall be logged.
10 [9:0]	RESERVED	—	—	—

#### 7.4.9 Interrupt Signal Enable (INTR\_SIGNAL\_ENABLE) (BASE + 0x28)

The Interrupt Signal Enable register enables or disables signaling of outstanding interrupts received by the Host Controller.

The bit fields in this register act as Interrupt Signal Enable Mask bits, according to the interrupt register model described in [Section 6.14](#). Writes to these fields may enable or disable the delivery of an interrupt condition to the Host (i.e. the Interrupt trigger).

**Table 25 Interrupt Signal Enable (INTR\_SIGNAL\_ENABLE) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
17 [31:15]	RESERVED	—	—	—
1 [14]	SCHED_CMD_MISSED_TICK_SIGNAL_EN	R/W	0x0	<b>Scheduled Command Missed Tick Signal Enable</b> When set to 1'b1 and the corresponding field in register <b>INTR_STATUS</b> is set, the Host Controller asserts an interrupt to the Host.
1 [13]	HC_ERR_CMD_SEQ_TIMEOUT_SIGNAL_EN	R/W	0x0	<b>Host Controller Command Sequence Stall/Timeout Signal Enable</b> When set to 1'b1 and the corresponding field in register <b>INTR_STATUS</b> is set, the Host Controller asserts an interrupt to the Host.
1 [12]	HC_WARN_CMD_SEQ_STALL_SIGNAL_EN	R/W	0x0	<b>Host Controller Internal Error Signal Enable</b> When set to 1'b1 and field <b>HC_SEQ_CANCEL_STAT</b> is set, the Host Controller asserts an interrupt to the Host.
1 [11]	HC_SEQ_CANCEL_SIGNAL_EN	R/W	0x0	<b>Host Controller Cancelled Transaction Sequence Signal Enable</b> When set to 1'b1 and field <b>HC_INTERNAL_ERR_STAT</b> is set, the Host Controller asserts an interrupt to the Host.
1 [10]	HC_INTERNAL_ERR_SIGNAL_EN	R/W	0x0	<b>Host Controller Internal Error Signal Enable</b> When set to 1'b1 and field <b>HC_INTERNAL_ERR_STAT</b> is set, the Host Controller asserts an interrupt to the Host.
10 [9:0]	RESERVED	—	—	—

#### 7.4.10 Interrupt Force (INTR\_FORCE) (BASE + 0x2C)

5633 The Interrupt Force register is used to force a specific interrupt. It can be used for debugging purposes.  
 5634 Writes to fields in this register act as Interrupt-force programming inputs, according to the interrupt register  
 5635 model described in *Section 6.14*.

5636

**Table 26 Interrupt Force (INTR\_FORCE) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
17 [31:15]	RESERVED	—	—	—
1 [14]	SCHED_CMD_MISSED_TICK_FORCE	W	0x0	For software testing, when set to 1'b1, forces the corresponding interrupt to be sent to Host, if the corresponding fields are set in registers <b>INTR_STATUS_ENABLE</b> and <b>INTR_SIGNAL_ENABLE</b> (see <i>Section 7.4.8</i> and <i>Section 7.4.9</i> ).
1 [13]	HC_ERR_CMD_SEQ_TIMEOUT_FORCE	W	0x0	
1 [12]	HC_WARN_CMD_SEQ_STALL_FORCE	W	0x0	
1 [11]	HC_SEQ_CANCEL_FORCE	W	0x0	
1 [10]	HC_INTERNAL_ERR_FORCE	W	0x0	
10 [9:0]	RESERVED	—	—	—

#### 7.4.11 Device Address Table Section Offset (DAT\_SECTION\_OFFSET) (BASE + 0x30)

5637 The Device Address Table Section Offset register holds the offset and size of the DAT table.

5638 **Table 27 Device Address Table Section Offset (DAT\_SECTION\_OFFSET) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
4 [31:28]	ENTRY_SIZE	R	IMPL	<b>DAT Entry size</b> Size of individual entry in the DAT table, in DWORDS. <b>Values:</b> <ul style="list-style-type: none"><li>• 0 (4'h0): DAT table entry size is 2 DWORDs</li><li>• 1–15: Reserved for future use</li></ul>
9 [27:19]	RESERVED	—	—	—
7 [18:12]	TABLE_SIZE	R	IMPL	<b>DAT Table Size</b> Size of the DAT, in entries. The total DAT table size is <b>ENTRY_SIZE * TABLE_SIZE</b> .
12 [11:0]	TABLE_OFFSET	R	IMPL	<b>DAT Table Offset</b> Offset of the DAT relative to the BASE Address of the current Bus Controller instance. If the Host Controller does not support implementing the DAT in registers (thereby forcing the software to provide the Device Context in memory), then it should indicate this by setting this field to 12'h000.

#### 7.4.12 Device Characteristics Table Section Offset (DCT\_SECTION\_OFFSET) (BASE + 0x34)

5639 The Device Characteristics Table Section Offset register holds the offset and size of the DCT table.

5640 **Table 28 Device Characteristics Table Section Offset (DCT\_SECTION\_OFFSET) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
4 [31:28]	ENTRY_SIZE	R	IMPL	<p><b>DCT Entry size</b> Size of individual entry in the DCT table, in DWORDS.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>0</b> (4'h0): DCT table entry size is 4 DWORDs</li> <li>• <b>1–15</b>: Reserved for future use.</li> </ul>
4 [27:24]	RESERVED	—	—	—
5 [23:19]	TABLE_INDEX	R/W	0x0	<p><b>DCT Table Index</b> Current index of the DCT, which is used as the starting index for the I3C <a href="#">ENTDAA</a> CCC. Once the complete Characteristics of the Device that won the Arbitration are written to the DCT (during <a href="#">ENTDAA</a> using Address Assignment Command) this index is incremented by 1. If needed, the software may override starting DCT index by setting this field.</p>
7 [18:12]	TABLE_SIZE	R	IMPL	<p><b>DCT Table Size</b> Size of the DCT, in entries. The total DCT table size is <b>ENTRY_SIZE * TABLE_SIZE</b>.</p>
12 [11:0]	TABLE_OFFSET	R	IMPL	<p><b>DCT Table Offset</b> Offset of the DCT relative to the BASE Address of the current Bus Controller instance. If the Host Controller does not support implementing the DCT in registers, thereby forcing the software to provide the Device Context, then it should indicate this by providing this field to 12'h000.</p>

#### 7.4.13 Ring Headers Section Offset (RING\_HEADERS\_SECTION\_OFFSET) (BASE + 0x38)

The Ring Headers Section Offset register indicates the location of the Ring Headers Section of the Register Map. If the Host Controller supports DMA Mode and implements a Ring Controller with at least one Ring Bundle, this register points to the offset of the start of the Ring Headers registers section (see *Section 7.6*).

Table 29 Ring Headers Section Offset (RING\_HEADERS\_SECTION\_OFFSET) Register

Size [Bits]	Field Name	Memory Access	Reset Value	Description
16 [31:16]	RESERVED	—	—	—
16 [15:0]	SECTION_OFFSET	R	IMPL	<b>Ring Headers Section Offset</b> Offset of the Ring Headers registers section of the Register Map, relative to the base offset of the current Bus Controller instance. A value of 0 in this register indicates that the Ring Headers section is not implemented.

#### 7.4.14 PIO Section Offset (PIO\_SECTION\_OFFSET) (BASE + 0x3C)

The PIO Section Offset register indicates the location of the PIO registers section of the Register Map. If the Host Controller supports PIO Mode and implements PIO registers, then this register points to the offset of the start of the PIO registers section (see *Section 7.5*).

Table 30 PIO Section Offset (PIO\_SECTION\_OFFSET) Register

Size [Bits]	Field Name	Memory Access	Reset Value	Description
16 [31:16]	RESERVED	—	—	—
16 [15:0]	SECTION_OFFSET	R	IMPL	<b>PIO Section Offset</b> Offset of the PIO registers section of the Register Map, relative to the base offset of the current Bus Controller instance. A value of 0 in this field indicates the PIO section is not implemented.

#### 7.4.15 Extended Capabilities Section Offset (EXT\_CAPS\_SECTION\_OFFSET) (BASE + 0x40)

The Extended Capabilities Section Offset register indicates the location of the Extended Capabilities section of the Register Map. If the Host Controller implements any Extended Capability structures, then this register points to the offset of the start of the first Extended Capability structure in the linked list of such structures (see *Section 7.7*).

**Table 31 Extended Capabilities Section Offset (EXT\_CAPS\_SECTION\_OFFSET) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
16 [31:16]	RESERVED	—	—	—
16 [15:0]	SECTION_OFFSET	R	IMPL	<b>Extended Capabilities Section Offset</b> Offset of the Extended Capabilities registers section of the Register Map, relative to the base offset of the current Bus Controller instance. A value of 0 in this field indicates that the Extended Capabilities section is not implemented.

#### 7.4.16 Internal Control Command Subtype Support (INT\_CTRL\_CMDS\_EN) (BASE + 0x4C)

This read-only register indicates the support for the various sub-commands of Internal Control Command type (see *Section 8.4.2*) that are supported by the Host Controller.

If a Host Controller supports a particular sub-command for the Internal Control Command (i.e., a particular value for field **MIPI\_CMD** for a Command Descriptor with field **CMD\_ATTR=0x7**), then the corresponding bit position in field **MIPI\_CMDS\_SUPPORTED** in this register shall be set to 1'b1. If the sub-command is not supported, then that bit position shall be set to 1'b0.

**Table 32 Internal Control Command Subtype Support (INT\_CTRL\_CMDS\_EN) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
16 [31:16]	RESERVED	R	0x0	–
15 [15:1]	MIPI_CMDS_SUPPORTED	R	IMPL	<p><b>MIPI Alliance Commands Supported</b>            The Host Controller provides a read-only bitmask of the Internal Control Command sub-commands that are supported (see <i>Section 8.4.2</i> and <i>Table 135</i>).            Each value of field <b>MIPI_CMD</b> greater than 0 (as defined in <i>Table 135</i>) that is supported for an Internal Control Command shall have the corresponding bit position set to a value of 1'b1.</p> <p><b>Examples:</b></p> <ul style="list-style-type: none"> <li>• Bit[4] shall be set to 1'b1, since the Target Reset Pattern Command (i.e., <b>MIPI_CMD=0x4</b>) is required to be supported.</li> <li>• Bit[5] shall be set to 1'b1, since at least some of the Controller SDA Recovery or Bus Reset Procedures (i.e., <b>MIPI_CMD=0x5</b>) are required to be supported.</li> </ul>
1 [0]	ICC_SUPPORT	R	0x1	<p><b>Internal Control Commands Supported</b>            Values:</p> <ul style="list-style-type: none"> <li>• <b>1'b1:</b> Some or all Internal Control Command sub-commands are supported.</li> <li>• <b>1'b0:</b> Illegal value, do not use.</li> </ul>

#### 7.4.17 IBI Notify Control (IBI\_NOTIFY\_CTRL) (BASE + 0x58)

5661

The IBI Notify Control register enables or disables event notifications for the IBI Queue/Ring.

5662

**Table 33 IBI Notify Control (IBI\_NOTIFY\_CTRL) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
<b>28</b> [31:4]	RESERVED	—	—	—
<b>1</b> [3]	NOTIFY_IBI_REJECTED	R/W	0x0	<p><b>Notify Rejected In-Band Interrupt Request Control</b></p> <p>Enables or disables reporting rejection of individual In-Band Interrupt Requests (IBIs). This field applies if an inbound IBI Request is NACKed and then IBIs are auto-disabled, based on either the <b>IBI_REJECT</b> field in the Target's DAT entry; or using the Target's credit counter, if the Target credit counting mechanism is supported and enabled (per <b>Section 6.9.5</b>).</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>1'b0: DISABLED:</b> Do not enqueue rejected IBI Status to the IBI Queue/Rings, if the incoming IBI is NACKed and is auto-disabled.</li> <li>• <b>1'b1: ENABLED:</b> Enqueue rejected IBI Status to the IBI Queue/Rings, if the incoming IBI is NACKed and is auto-disabled. If the Target IBI credit counting mechanism is supported and currently enabled, then the Host Controller shall only send one such rejected IBI Status per Target, while that Target's credit counter remains at zero.</li> </ul>
<b>1</b> [2]	RESERVED	—	—	—
<b>1</b> [1]	NOTIFY_CRR_REJECTED	R/W	0x0	<p><b>Notify Rejected Controller Role Request Control</b></p> <p>Enables or disables reporting rejection of individual Controller Role Requests.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>1'b0: DISABLED:</b> Do not pass rejected IBI Status to IBI Queue/Ring, if the incoming Controller Role Request is NACKed and is auto-disabled based on the <b>CRR_REJECT</b> field in relevant DAT entry.</li> <li>• <b>1'b1: ENABLED:</b> Pass rejected IBI Status to the IBI Queue, if the incoming Controller Role Request is NACKed and is auto-disabled based on the <b>CRR_REJECT</b> field in the relevant DAT entry.</li> </ul>

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [0]	NOTIFY_HJ_REJECTED	R/W	0x0	<p><b>Notify Rejected Hot-Join Control</b> Enables or disables reporting rejection of individual Hot Join requests.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>1'b0: DISABLED:</b> Do not pass rejected IBI Status to IBI Queue, if the incoming Hot-Join request is NACKed and is auto-disabled based on field <b>HOT_JOIN_CTRL</b> of register <b>HC_CONTROL</b>.</li> <li>• <b>1'b1: ENABLED:</b> Pass rejected IBI Status to the IBI Queue, if the incoming Hot Join request is NACKed and is auto-disabled based on field <b>HOT_JOIN_CTRL</b> of register <b>HC_CONTROL</b>.</li> </ul>

#### 7.4.18 IBI Data Abort Control (IBI\_DATA\_ABORT\_CTRL) (BASE + 0x5C)

The IBI Data Abort Control register allows the Driver to configure the Host Controller's monitoring logic for ongoing IBI data payload transfers. When enabled, the Driver may request the I3C Bus Controller Logic to abort (i.e., to forcibly stop) an ongoing In-Band Interrupt Request, if and when the Driver determines that the IBI Data Payload should no longer be allowed to continue (see [Section 6.9.4](#)).

5667

**Table 34 IBI Data Abort Control (IBI\_DATA\_ABORT\_CTRL) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [31]	IBI_DATA_ABORT_MON	R/W	0x0	<p><b>IBI Data Abort Monitor</b> Enables or disables the monitoring logic that allows the Driver to request an IBI Data Abort operation.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: <b>DISABLED</b>: No IBI Data Abort operations may be requested by the Driver.</li> <li>• 1'b1: <b>ENABLED</b>: Driver may request an IBI Data Abort operation; Host Controller shall monitor successive writes to this register.</li> </ul>
10 [30:21]	RESERVED	—	—	—
3 [20:18]	MATCH_STATUS_TYPE	W	0x0	<p><b>Match IBI Status Type</b> The Host shall provide the value of field STATUS_TYPE from the first IBI Status Descriptor sent for this IBI data payload.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 3'b000: <b>REGULAR_IBI</b>: If matching, this aborts an IBI data payload for a regular IBI.</li> <li>• 3'b100: <b>AUTOCMD_READ</b>: If matching, this aborts an IBI data payload for an Auto-Command (i.e., the Pending Read Data portion).</li> <li>• <b>Other Values</b>: Not supported, do not use.</li> </ul>
2 [17:16]	AFTER_N_CHUNKS	W	0x0	<p><b>Abort After N Chunks</b> When an IBI Data Abort operation is requested by the Driver, this field determines how many additional data Chunks will be allowed before the I3C Bus Controller Logic forcibly ends the data transfer (e.g., using the T-bit, per <a href="#">Section 5.1.2.3.4</a> of Version 1.1.1+ of the I3C Specification [<a href="#">MIP105</a>]).</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 2'd0: <b>IMMED</b>: Abort the IBI data payload transfer immediately (i.e., 0 more Chunks).</li> <li>• 2'd1–2'd3 (N): <b>DELAYED</b>: Abort the IBI data payload transfer after N more Chunks (i.e., N from 1 thru 3). If the Target terminates the IBI data payload before this count is reached, then no abort is performed.</li> </ul>

Size [Bits]	Field Name	Memory Access	Reset Value	Description
8 [15:8]	MATCH_IBI_ID	W	0x0	<p><b>Match IBI Target Address</b>            The Host shall provide the Target's Address for the current (i.e., ongoing) IBI data payload.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>Bits[15:9]:</b> The Target's Device Address</li> <li>• <b>Bit[8]:</b> 1'b1 always (i.e., RnW bit) for an IBI Data Abort operation request</li> </ul>
8 [7:0]	RESERVED	—	—	—

5668 **Note:**

5669     *This register is new for this version of this I3C HCI Specification. The Driver can determine whether*  
 5670     *the Host Controller supports the IBI Data Abort operation by reading field **IBI\_DATA\_ABORT\_EN** in*  
 5671     *register **HC\_CAPABILITIES** (see **Section 7.4.4**).*

5672     *If the capability is not supported, then this register will not be present, and the Host Controller will*  
 5673     *return a value of all zeros. This allows for backwards compatibility with previous versions of this I3C*  
 5674     *HCI Specification.*

5675     *If this capability is supported and enabled, then the Driver shall monitor new IBI Status Descriptors*  
 5676     *and shall determine when to request the IBI Data Abort operation (if needed). The Driver shall*  
 5677     *provide matching values of fields **STATUS\_TYPE** and **IBI\_ID** from the first IBI Status Descriptor. If*  
 5678     *these values do not match the corresponding fields in this register, or if the IBI data payload has*  
 5679     *already finished by the time the Host Controller receives the write to this register, then no IBI Data*  
 5680     *Abort operation shall be performed. Additionally, writes to these match fields are not ‘sticky’ and will*  
 5681     *not apply to any subsequent IBIs from the same Target.*

5682     *If the Host writes to this register with Bit[8] set to 1'b0, then values written to other fields (except*  
 5683     *field **IBI\_DATA\_ABORT\_EN**) shall be ignored.*

#### 7.4.19 Device Context Base Address Low (DEV\_CTX\_BASE\_LO) (BASE + 0x60)

The Device Context Base Address Low register points to Driver-allocated memory containing Device Context (i.e., DAT and/or DCT) when either or both are provided in Host system memory (per [Section 7.3.3](#)).

This register and the following one (i.e., [DEV\\_CTX\\_BASE\\_HI](#)) are only present for Host Controllers that support direct access to Host system memory through a Memory Access Engine (i.e., DMA engine). If a Host Controller does not support direct access to Host system memory, then these registers shall be read-only, and both the DAT and the DCT shall be implemented in registers.

If the Host Controller also supports Scatter-Gather capability (as indicated in field [SG\\_CAPABILITY\\_DC\\_EN](#) of register [HC\\_CAPABILITIES](#); see [Section 7.4.4](#)), then the appropriate Scatter-Gather register shall also be implemented as register [DEV\\_CTX\\_SG](#) (see [Section 7.4.21](#)).

If either the DAT Section Offset (i.e., register [DAT\\_SECTION\\_OFFSET](#), see [Section 7.4.11](#)) or the DCT Section Offset (i.e., register [DCT\\_SECTION\\_OFFSET](#), see [Section 7.4.12](#)) indicate that the Host Controller does not implement table entries in registers (with field [TABLE\\_OFFSET](#) in either register having a value of 12'h000), then the Driver must allocate enough System memory to contain the required number of entries. The total amount of memory for each section shall be the sum of the number of entries multiplied by the size of the entry. If both the DAT and the DCT are required, then the total size of allocated memory can be expressed as follows:

$$\text{sizeof ( DAT )} + \text{sizeof ( DCT )}$$

- If Scatter-Gather mode is not supported, then this must be a single contiguous allocation of physical memory, large enough to hold all table entries.
  - The Driver shall write the base address of this single allocation of memory into the Base Address registers.
- If Scatter-Gather mode is supported, then this might be either a single contiguous allocation (as above), or it might consist of a series of individual chunks (i.e., smaller allocations) having a total size sufficient to hold all table entries described by an array of Memory Descriptor structures (see [Section 8.7](#)).
  - The Driver shall set field [BLP](#) of register [DEV\\_CTX\\_SG](#) to indicate whether it is providing a single allocation (as above) or an array of individual allocations.
  - For an array of individual allocations, the Driver shall write the address pointer for the array of Memory Descriptor structures into the Base Address registers, and shall write the size of this array into field [LIST\\_SIZE](#) of register [DEV\\_CTX\\_SG](#).
  - Refer to [Section 6.2.1](#) for more information on Scatter-Gather memory allocation.

**Table 35 Device Context Address Low (DEV\_CTX\_BASE\_LO) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	BASE_LO	R/W	0x0	<b>Device Context Base Low</b> Lower 32 bits of pointer to physical memory allocated for storing Device Context. The Device Context is DWORD aligned, so the last two Address bits will always be 2'b00.

#### 7.4.20 Device Context Base Address High (DEV\_CTX\_BASE\_HI) (BASE + 0x64)

The Device Context Base Address High register points to Driver-allocated memory containing Device Context (i.e., DAT and/or DCT) when either or both are provided in Host system memory.

This register and the preceding one (i.e., **DEV\_CTX\_BASE\_LO**) are only present for Host Controllers that support direct access to Host system memory through a Memory Access Engine (i.e., DMA engine). If a Host Controller does not support direct access to Host system memory, then these registers shall be read-only, and both the DAT and the DCT shall be implemented in registers.

Refer to the details for the preceding register **DEV\_CTX\_BASE\_LO** for requirements and conditions for use.

**Table 36 Device Context Base Address High (DEV\_CTX\_BASE\_HI) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	BASE_HI	R/W	0x0	<b>Device Context Base High</b> Upper 32 bits of pointer to physical memory allocated for storing Device Context.

#### 7.4.21 Device Context Scatter-Gather Support (DEV\_CTX\_SG) (BASE + 0x68)

The Device Context Scatter-Gather Support register allows Driver-provided memory for Device Context tables (i.e., DAT and/or DCT) to be allocated as multiple chunks of physically contiguous memory, using Scatter-Gather allocation per *Section 6.2.1*.

This register is only present for Host Controllers that support direct access to Host system memory, with the Base Address registers (i.e., **DEV\_CTX\_BASE\_LO** and **DEV\_CTX\_BASE\_HI**), and that also support Scatter-Gather capability, indicated with field **SG\_CAPABILITY\_DC\_EN** of register **HC\_CAPABILITIES** (see *Section 7.4.4*). If a Host Controller does not support Scatter-Gather capability, then this register shall be read-only.

**Table 37 Device Context Scatter-Gather Support (DEV\_CTX\_SG) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [31]	BLP	R/W	0x0	<b>Buffer Vs. List Pointer</b> Indicates whether the Base Address registers <b>DEV_CTX_BASE_HI</b> and <b>DEV_CTX_BASE_LO</b> points directly to the Buffer, or to a List of Memory Descriptors describing the Buffer. <b>Values:</b> <ul style="list-style-type: none"> <li>• 1'b0: BUFFER: Pointer points to a single allocation of physical memory</li> <li>• 1'b1: LIST: Pointer points to physical memory comprising an array of Memory Descriptors (i.e., Scatter-Gather Buffers that are not physically continuous)</li> </ul>
15 [30:16]	RESERVED	R/W	0x0	–
16 [15:0]	LIST_SIZE	R/W	0x0	<b>List Size</b> List Size in Entries

## 7.5 PIO Mode Registers

All PIO Mode registers are defined as relative offsets, starting from the location indicated by register **PIO\_SECTION\_OFFSET** (see *Section 7.4.14*), if PIO Mode is supported.

### 7.5.1 Command Queue Port (**COMMAND\_QUEUE\_PORT**) (PIO + 0x0)

32-bit mailbox register **COMMAND\_QUEUE\_PORT** is a write-only register that the Driver uses to enqueue a Command Descriptor structure in a format that depends on the requested command type. All command types are defined in *Section 8.4*.

When writing to this register to enqueue a Command Descriptor structure, the Driver must write DWORDs in the correct order, starting with the Least Significant DWORD (i.e., starting at bit 0) until the Most Significant DWORD. The Driver shall use this register to enqueue new commands, including Transfer Commands, per *Section 6.8.1*.

For a 64-bit Command Descriptor, the two DWORDs are written in order, from lowest to highest (i.e., DWORD 0 followed by DWORD 1).

Upon receiving two DWORD writes to register **COMMAND\_QUEUE\_PORT**, the Host Controller shall enqueue the new Command Descriptor.

**Note:**

*Usage of a Command Descriptor is defined in **Section 6.7**.*

### 7.5.2 Response Queue Port (**RESPONSE\_QUEUE\_PORT**) (PIO + 0x4)

32-bit mailbox register **RESPONSE\_QUEUE\_PORT** is a read-only register that the Driver uses to read a Response Descriptor structure (see *Section 8.5*) generated in the Response Queue, after a command that was previously processed by the Host Controller.

Each DWORD read of this register shall consume (i.e., shall dequeue) one Response Descriptor from the Response Queue. The Driver shall read from this register to determine the status of commands, such as Transfer Commands, per *Section 6.8.1*.

The Driver should not attempt to read from this register unless there is at least one Response Descriptor pending in the Response Queue. The Driver may receive a notification based on Queue Status indication, in the form of a PIO Interrupt (i.e., field **RESP\_READY\_STAT** in register **PIO\_INTR\_STATUS**; see *Section 7.5.9*).

### 7.5.3 Transfer Data Port (XFER\_DATA\_PORT) (PIO + 0x8)

32-bit mailbox register **XFER\_DATA\_PORT** is a 32-bit bi-directional data transfer register that is used both to read from the **RX\_DATA\_PORT** (which reads from the Rx Data Queue), and to write to the **TX\_DATA\_PORT** (which writes to the Tx Data Queue). In other words, registers **RX\_DATA\_PORT** and **TX\_DATA\_PORT** have the same offset, forming a single bi-directional port for transmitting or receiving Host Controller data for Transfer Commands per *Section 6.8.1*.

- **Read Operations:** To receive data from the RX Buffer, the Driver shall read from register **RX\_DATA\_PORT**. Its reset value is undefined, and should be read based on Queue Status indication. The Host Controller may send a notification in the form of a PIO Interrupt.
- **Write Operations:** To send data to the TX Buffer, the Driver shall write to register **TX\_DATA\_PORT**.

**Note:**

*The ordering of bytes within the DWORDs that are written into the Tx Data Queue or read from the Rx Data Queue via this register shall use the current data byte ordering mode, per Section 6.8.3.*

Table 38 Rx Data Port (RX\_DATA\_PORT) Register

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	RX_DATA	R	0x0	<p><b>Data Read from the Rx Data Buffer</b></p> <p>The Rx Data Port is mapped to the Rx Data Buffer. The Receive data is always aligned to a 4-byte boundary, and stored in the Rx Data Buffer. If the length of the data transfer is not aligned to a 4-byte boundary, then there will be extra (unused) bytes at the end of the transferred data. The valid data must be identified using the <b>DATA_LENGTH</b> field in the Response Descriptor.</p>

Table 39 Tx Data Port (TX\_DATA\_PORT) Register

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	TX_DATA	W	0x0	<p><b>Data Written to the Tx Data Buffer</b></p> <p>The Tx Data Port is mapped to the Tx Data Buffer. The transmit data should always start aligned to a 4-byte boundary, and written to the Transmit Data Port register. If the length of the transfer is not aligned to a 4-byte boundary, then there will be extra (unused) bytes at the end of the transferred data. The Controller shall only send the valid number of bytes indicated in the <b>DATA_LENGTH</b> field of the Command Descriptor.</p>

### 7.5.4 IBI Port (IBI\_PORT) (PIO + 0xC)

5772 32-bit mailbox register **IBI\_PORT** is a read-only register the Driver uses to read DWORDs from the IBI  
 5773 Queue containing an IBI Status Descriptor (see *Section 8.6*) and any associated IBI Data, after an Interrupt  
 5774 Request (e.g., an In-Band Interrupt Request) was received by the Host Controller.

5775 When receiving an IBI, the Driver shall first read the IBI Status Descriptor, followed by consecutive  
 5776 DWORDs of IBI Data (as specified in *Section 6.9.1*). The Host Controller shall enqueue the IBI Status  
 5777 Descriptor as the first DWORD for the IBI event, followed by any DWORDs of data associated with that  
 5778 IBI event.

5779 **Note:**

5780 *If the I3C HCI Auto-Command feature is used, then the IBI Data will include the data received via*  
 5781 *the auto-generated private read operation.*

5782 The Driver should not attempt to read from this register unless there is at least one IBI Status Descriptor  
 5783 pending in the IBI Data Queue. The Driver may receive a notification based on Queue Status indication, in  
 5784 the form of a PIO Interrupt (i.e., field **IBI\_STATUS\_THLD\_STAT** in register **PIO\_INTR\_STATUS**; see  
 5785 *Section 7.5.9*).

5786 **Table 40 IBI Port (IBI\_PORT) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	IBI_DATA	R	0x0	<p><b>Data Read from IBI Data Buffer</b></p> <p>The IBI Port is mapped to the IBI Data Queue. IBI Data is always aligned to a 4-byte boundary and then put into the IBI Queue.</p> <p>If the incoming data is not aligned to a 4-byte boundary, then there will be extra (unused) bytes at the end of the transferred IBI data. This can be determined from the value of field <b>DATA_LENGTH</b> in the IBI Status Descriptor.</p>

### 7.5.5 Queue Threshold Control (QUEUE\_THLD\_CTRL) (PIO + 0x10)

The Queue Threshold Control register controls the interrupt trigger thresholds for the Command Queue, the Response Queue, and the IBI Queue. It also controls the maximum size of the IBI Data segment for each IBI Status Descriptor, for IBIs that have long payload data and require multiple IBI Status Descriptors (per [Section 6.9.1](#)).

**Note:**

*It is assumed that the Host Controller has exactly one Command Queue, exactly one Response Queue, and exactly one IBI Queue.*

The specific reset values are indicative, and could be hardware implementation specific.

**Table 41 Queue Threshold Control (QUEUE\_THLD\_CTRL) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
8 [31:24]	IBI_STATUS_THLD	R/W	0x1	<p><b>IBI Status Threshold</b>  Controls generation of the <b>IBI_STATUS_THLD_STAT</b> interrupt, based on the value of the IBI Queue's outstanding IBI Statuses.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>0:</b> Not valid, do not use</li> <li>• <b>1–255 (N):</b> Interrupt is generated when the number of outstanding IBI Statuses is equal to or greater than <b>N</b>.</li> </ul> <p>Each IBI Status entry can represent either the complete IBI payload (if the IBI payload byte size <math>\leq 4 * \text{IBI\_DATA\_SEGMENT\_SIZE}</math>), or a segment of the IBI payload (if the IBI payload byte size <math>&gt; 4 * \text{IBI\_DATA\_SEGMENT\_SIZE}</math>).</p>
8 [23:16]	IBI_DATA_SEGMENT_SIZE	R/W	0x1	<p><b>IBI Data Segment Size</b>  Controls the maximum size of IBI Data segments, in DWORDs.</p> <p><b>Supported Values:</b></p> <ul style="list-style-type: none"> <li>• <b>Minimum:</b> 1 (1 DWORD = 4 Bytes)</li> <li>• <b>Maximum:</b> 63 (63 DWORDs = 252 Bytes), provided that the configured IBI Queue depth is <math>\geq 64</math>.</li> </ul> <p>In PIO Mode, this field allows the incoming IBI Data to be sliced into multiple segments generating status individually, to support cut-through readout of a long IBI Payload Data. When Asynchronous Timing Control Mode is supported, this field should be set to a value of 4 or more to allow the single data segment to contain the entire Controller timestamp value (i.e., both <b>C_REF</b> and <b>C_C2</b>).</p>

Size [Bits]	Field Name	Memory Access	Reset Value	Description
8 [15:8]	<b>RESP_BUF_THLD</b>	R/W	0x1	<p><b>Response Ready Buffer Threshold</b>  Controls the minimum number of Response Queue entries needed to trigger the <b>RESP_READY_STAT</b> interrupt.  If this field is greater than the value reported by the Response Queue Size, then the full buffer depth will be considered (see Note below).</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>0:</b> Not valid, do not use</li> <li>• <b>1–255 (N):</b> Interrupt is triggered when Response Queue contains at least N entries</li> </ul>
8 [7:0]	<b>CMD_EMPTY_BUF_THLD</b>	R/W	0x1	<p><b>Command Ready Buffer Threshold</b>  Controls the minimum number of empty Command Queue entries needed to trigger the <b>CMD_QUEUE_READY_STAT</b> interrupt.  If this field is greater than (<b>QUEUE_SIZE.CR_QUEUE_SIZE</b> – 1), then the full buffer depth will be considered.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>0:</b> Not valid, do not use</li> <li>• <b>1–255 (N):</b> Interrupt is issued when Command Queue contains at least N empty entries</li> </ul>

5796 **Note:**

5797 In version 1.1 of the HCI Specification, the Command Queue and Response Queue have the same  
5798 size. In this version of the HCI Specification, the Response Queue might have a different size, as  
5799 indicated by register **ALT\_QUEUE\_SIZE** (per **Section 7.5.8**, if that register is present and indicates  
5800 this status).

5801 If the Response Queue size is indicated by register **ALT\_QUEUE\_SIZE** (i.e., where it might be  
5802 different from the Command Queue size), then the highest possible value of N for field  
5803 **RESP\_BUF\_THLD** shall be (**ALT\_QUEUE\_SIZE.ALT\_RESP\_QUEUE\_SIZE** – 1).

5804 If register **ALT\_QUEUE\_SIZE** is not present, or if it is present but indicates that the Response Queue  
5805 size is the same as the Command Queue size, then the highest possible value of N for field  
5806 **RESP\_BUF\_THLD** shall be (**QUEUE\_SIZE.CR\_QUEUE\_SIZE** – 1).

### 7.5.6 Transfer Data Buffer Threshold Control (DATA\_BUFFER\_THLD\_CTRL) (PIO + 0x14)

The Data Buffer Control register controls the interrupt trigger thresholds for the Rx Data Buffer Queue and the Tx Data Buffer Queue. This register also controls when the Host Controller will start an enqueued Write Transfer or a Read Transfer, based on the amount of data in the Tx Data Buffer or the available (i.e., empty) entries in the Rx Data Buffer.

**Table 42 Data Buffer Threshold Control (DATA\_BUFFER\_THLD\_CTRL) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
5 [31:27]	RESERVED	-	-	-
3 [26:24]	RX_START_THLD	R/W	0x1	<p><b>Receive Start Threshold in DWORDs</b>  When preparing to initiate a Read Transfer on the I3C Bus, the Host Controller shall wait until the Receive Buffer has at least the indicated number of DWORD entries available to receive the data bytes (see <a href="#">Section 6.5</a>).</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 0x0–0x7 (N): Wait until <math>(2 ^ (N+1))</math> DWORD entries are available</li> </ul> <p><b>Examples:</b></p> <ul style="list-style-type: none"> <li>• 0x0: Wait until 2 DWORD entries are available</li> <li>• 0x7: Wait until 256 DWORD entries are available</li> </ul>
5 [23:19]	RESERVED	-	-	-
3 [18:16]	TX_START_THLD	R/W	0x1	<p><b>Transmit (Transfer) Start Threshold in DWORDs</b>  When preparing to initiate a Write Transfer on the I3C Bus, the Host Controller shall wait until the Driver has written at least the indicated number of entries (i.e., DWORDs) into the Transmit Buffer. See <a href="#">Section 6.5.5</a>.  For Write-type transfers that only use Immediate Data, the Host Controller shall not wait for DWORDs to be written into the Transmit Buffer. In such cases, this Transmit Start Threshold is bypassed.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 0x0–0x7 (N): Wait for at least <math>(2 ^ (N+1))</math> DWORDs to be written</li> </ul> <p><b>Examples:</b></p> <ul style="list-style-type: none"> <li>• 0x0: Wait for at least 2 DWORDs to be written</li> <li>• 0x7: Wait for at least 256 DWORDs to be written</li> </ul>
5 [15:11]	RESERVED	-	-	-

Size [Bits]	Field Name	Memory Access	Reset Value	Description
3 [10:8]	RX_BUF_THLD	R/W	0x1	<p><b>Receive Buffer Threshold</b>            Minimum number of Receive FIFO entries of data received, in DWORDs, that will trigger the <b>RX_THLD_STAT</b> interrupt (see <b>Section 6.5.5</b> and <b>Section 6.8.1</b>).  <b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>0x0–0x7 (N):</b> Interrupt triggers when <math>(2^N + 1)</math> Rx Buffer DWORD entries are received during the Read transfer</li> </ul> <p><b>Examples:</b></p> <ul style="list-style-type: none"> <li>• <b>0x0:</b> Interrupt triggers at 2 Rx Buffer DWORD entries</li> <li>• <b>0x7:</b> Interrupt triggers at 256 Rx Buffer DWORD entries</li> </ul> <p><b>Note:</b>  <i>Since the minimum threshold value is 2 Rx Buffer entries, the Driver shall be responsible for reading the last DWORD of data for the Read transfer, based on the value of field <b>DATA_LENGTH</b> in the Response Descriptor (see <b>Section 6.8.1</b> for additional details).</i></p>
5 [7:3]	RESERVED	—	—	—
3 [2:0]	TX_BUF_THLD	R/W	0x1	<p><b>Transmit Buffer Threshold</b>            Minimum number of available Transmit FIFO entries, in DWORDs, that will trigger the <b>TX_THLD_STAT</b> interrupt (see <b>Section 6.5.5</b> and <b>Section 6.8.1</b>).  <b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>0x0–0x7 (N):</b> Interrupt triggers when <math>(2^N + 1)</math> Tx Buffer DWORD entries are available</li> </ul> <p><b>Examples:</b></p> <ul style="list-style-type: none"> <li>• <b>0x0:</b> Interrupt triggers at 2 Tx Buffer DWORD entries available</li> <li>• <b>0x7:</b> Interrupt triggers at 256 Tx Buffer DWORD entries available</li> </ul>

5812 **Note:**

5813 For fields **RX\_START\_THLD** and **RX\_BUF\_THLD**, the Driver must provide a value less than the size  
 5814 of the Rx Data Queue, as indicated by field **RX\_DATA\_BUFFER\_SIZE** in register **QUEUE\_SIZE**.

5815 For fields **TX\_START\_THLD** and **TX\_BUF\_THLD**, the Driver must provide a value less than or equal  
 5816 to the size of the Tx Data Queue, as indicated by field **TX\_DATA\_BUFFER\_SIZE** in register  
 5817 **QUEUE\_SIZE**.

5818 In this version of the I3C HCI Specification, the threshold values have the same meanings as  
 5819 versions 1.0 and 1.1 of the I3C HCI Specification. For clarity and brevity, the field descriptions now  
 5820 use formulas.

### 7.5.7 Queue Size (QUEUE\_SIZE) (PIO + 0x18)

5821 This register contains read-only parameters that describe the PIO Queues, including the Command Queue,  
5822 Response Queue, Data buffer, and IBI status queue sizes. Values defined in this register determine the  
5823 allowed values for the PIO Queue Thresholds.

5824 **Table 43 Queue Size (QUEUE\_SIZE) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
8 [31:24]	TX_DATA_BUFFER_SIZE	R	IMPL	<p><b>Transmit Data Buffer Size</b> In DWORDs</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 0x0: 2 DWORDs</li> <li>• 0x1: 4 DWORDs</li> <li>• 0x2: 8 DWORDs</li> <li>• 0x3: 16 DWORDs</li> <li>• 0x4: 32 DWORDs</li> <li>• 0x5: 64 DWORDs</li> <li>• 0x6: 128 DWORDs</li> <li>• 0x7: 256 DWORDs</li> <li>• 0x8–0xF: Reserved for future use</li> </ul>
8 [23:16]	RX_DATA_BUFFER_SIZE	R	IMPL	<p><b>Receive Data Buffer Size</b> In DWORDs</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 0x0: 2 DWORDs</li> <li>• 0x1: 4 DWORDs</li> <li>• 0x2: 8 DWORDs</li> <li>• 0x3: 16 DWORDs</li> <li>• 0x4: 32 DWORDs</li> <li>• 0x5: 64 DWORDs</li> <li>• 0x6: 128 DWORDs</li> <li>• 0x7: 256 DWORDs</li> <li>• 0x8–0xF: Reserved for future use</li> </ul>
8 [15:8]	IBI_STATUS_SIZE	R	IMPL	<p><b>IBI Queue Size</b> Size of IBI queue (see Note below).</p> <p><b>Values:</b> 2–255 (N)</p> <ul style="list-style-type: none"> <li>• If field EXT_IBI_QUEUE_EN = 1'b0, then this value (N) indicates the exact size in DWORDs.</li> <li>• If field EXT_IBI_QUEUE_EN = 1'b1, then the IBI Queue has extended size, and this size is exactly (8 * N) DWORDs.</li> </ul> <p><b>Note:</b> <i>The size of the IBI Queue also helps to determine the maximum valid index for programming the IBI Status Threshold, via field IBI_STATUS_THLD in register QUEUE_THLD_CTRL.</i></p>

Size [Bits]	Field Name	Memory Access	Reset Value	Description
8 [7:0]	CR_QUEUE_SIZE	R	IMPL	<p><b>Command/Response Queue Size</b>            Size of Command and Response queues, in entries (see Note below).            The Command Descriptor size is two DWORDs (per <b>Section 8.4</b>), and the Response Descriptor size is one DWORD (per <b>Section 8.5</b>).  <b>Values:</b> 2–255 entries</p>

5825 **Note:**

5826 If register **ALT\_QUEUE\_SIZE** is present (**Section 7.5.8**), and if that register indicates that it defines  
 5827 the size of the Response Queue (i.e., if field **ALT\_RESP\_QUEUE\_EN** is 1'b1), then field  
 5828 **CR\_QUEUE\_SIZE** in this register only defines the size of the Command Queue. In that case, the  
 5829 Command Queue and Response Queue might have different sizes.

5830 If register **ALT\_QUEUE\_SIZE** is present, and if that register indicates that the IBI Queue Size is  
 5831 extended (i.e., if field **EXT\_IBI\_QUEUE\_EN** is 1'b1), then the true size of the IBI Queue (in DWORDs)  
 5832 shall be the value of field **IBI\_STATUS\_SIZE** multiplied by 8.

### 7.5.8 Alternate Queue Size (ALT\_QUEUE\_SIZE) (PIO + 0x1C)

This register contains additional read-only parameters that describe the PIO Queues, including:

- The Response Queue, if it does not have the same size as the Command Queue; and
- The IBI Queue, if it is extended (i.e., is larger than the size reported in field **IBI\_STATUS\_SIZE** in register **QUEUE\_SIZE**, see [Section 7.5.7](#)).

This register is conditionally required if either the Response Queue size is defined separately from the Command Queue size, or if the IBI Queue Size is extended (i.e., is 8x the value reported in field **IBI\_STATUS\_SIZE**).

**Table 44 Alternate Queue Size (ALT\_QUEUE\_SIZE) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
3 [31:29]	RESERVED	-	-	-
1 [28]	EXT_IBI_QUEUE_EN	R	IMPL	<b>Extended IBI Queue Size</b> If set to 1'b1, this register indicates that the IBI Queue size is extended. The true size of the IBI Queue is exactly 8 DWORDs multiplied by the value reported in field <b>IBI_STATUS_SIZE</b> . The valid size of the IBI Queue can therefore range from 16–2040 DWORDs. If set to 1'b0, then the IBI Queue size is not extended, and the true size of the IBI Queue is exactly the number of DWORDs reported in field <b>IBI_STATUS_SIZE</b> . The valid size of the IBI Queue can therefore range from 2–255 DWORDs.
3 [27:25]	RESERVED	-	-	-
1 [24]	ALT_RESP_QUEUE_EN	R	IMPL	<b>Alternate Response Queue</b> If set to 1'b1, then this register defines the size of the Response Queue, separately from the size of the Command Queue. If set to 1'b0, then this register does not define the Response Queue size, i.e., the Command and Response Queues have the same size, per field <b>CR_QUEUE_SIZE</b> in register <b>QUEUE_SIZE</b> ( <a href="#">Section 7.5.7</a> ).
16 [23:8]	RESERVED	-	-	-
8 [7:0]	ALT_RESP_QUEUE_SIZE	R	IMPL	<b>Alternate Response Queue Size</b> Size of Response Queues, in entries. Only valid if field <b>ALT_RESP_QUEUE_EN</b> is set to 1'b1 (see Note below). The Response Descriptor size is one DWORD, per <a href="#">Section 8.5</a> . <b>Values:</b> 2–255 entries

5841

**Note:**

5842

This register is new for this version of this I3C HCI Specification. The Driver shall try to read this register when initializing the Host Controller, to determine whether the Command Queue and Response Queue have different sizes, and whether the IBI Queue size is extended. In previous versions of this I3C HCI Specification, the Command Queue and Response Queue were required to have the same size, and the maximum IBI Queue size was limited to 255 DWORDs.

5843

If the IBI Queue size is extended, then this shall have no effect on how the Host Controller treats the value written to field **IBI\_STATUS\_THLD** in register **QUEUE\_THLD\_CTRL**.

5844

If this register is not present, or if the Host Controller returns a value of all zeros, then the Driver should assume that none of the additional options are supported, namely: the size of the Response Queue is the same as the size of the Command Queue, as reported by field **CR\_QUEUE\_SIZE** in register **QUEUE\_SIZE**, and the IBI Queue size is not extended (i.e., is exactly the number of DWORDs reported by field **IBI\_STATUS\_SIZE**). This ensures that Drivers can remain compatible with existing implementations that conform to previous versions of this I3C HCI Specification. Note that such implementations will return values of all zeros for registers that are not present.

5845

5846

5847

5848

5849

5850

5851

5852

5853

5854

5855

### 7.5.9 PIO Interrupt Status (PIO\_INTR\_STATUS) (PIO + 0x20)

The PIO Interrupt Status register indicates the status of outstanding interrupts relating to the PIO Queues or other PIO events and errors.

The status fields are either R/W1C (write 1 to clear), or else are cleared based on queue operations.

The bit fields in this register act as Status Bits, according to the interrupt register model described in [Section 6.14](#). The Host Controller shall set these bits (i.e., the “Status bit set” action will occur) if the interrupt condition is detected, and if the corresponding bit in the Interrupt Status Enable Mask is set to 1'b1.

**Table 45 PIO Interrupt Status (PIO\_INTR\_STATUS) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
<b>22</b> [31:10]	RESERVED	—	—	—
<b>1</b> [9]	TRANSFER_ERR_STAT	R/W1C	0x0	<b>Transfer Error Status</b> The Host Controller shall set this bit to 1'b1 when any transfer error occurs on the I3C Bus. The Error Type for this error shall be available in the Response structure corresponding to this transfer/command.
<b>3</b> [8:6]	RESERVED	—	—	—
<b>1</b> [5]	TRANSFER_ABORT_STAT	R/W1C	0x0	<b>Transfer Abort Status</b> The Host Controller shall set this bit to 1'b1 when any transfer is aborted.
<b>1</b> [4]	RESP_READY_STAT	R	0x0	<b>Response Ready Status</b> The Host Controller shall set this bit to 1'b1 when the number of Response Queue entries is $\geq$ the <b>RESP_BUF_THLD</b> threshold (see register <a href="#">QUEUE_THLD_CTRL</a> ). The Host Controller shall automatically clear this field to 1'b0 when the number of Response Queue entries falls below the <b>RESP_BUF_THLD</b> threshold.
<b>1</b> [3]	CMD_QUEUE_READY_STAT	R	0x0	<b>Command Queue Ready Status</b> The Host Controller shall set this bit to 1'b1 when the number of empty Command Queue entries is $\geq$ the <b>CMD_EMPTY_BUF_THLD</b> threshold (see register <a href="#">QUEUE_THLD_CTRL</a> ). The Host Controller shall automatically clear this field to 1'b0 when the number of empty Command Queue entries falls below the <b>CMD_EMPTY_BUF_THLD</b> threshold.

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [2]	IBI_STATUS_THLD_STAT	R	0x0	<p><b>IBI Status Threshold Status</b></p> <p>The Host Controller shall set this bit to 1'b1 when the number of IBI Status Entries in the IBI Queue reaches the <b>IBI_STATUS_THLD</b> threshold (see register <b>QUEUE_THLD_CTRL</b>).</p> <p>The Host Controller shall automatically clear this field to 1'b0 when the number of IBI Status entries in the IBI Queue falls below the <b>IBI_STATUS_THLD</b> threshold as a result of application reads.</p>
1 [1]	RX_THLD_STAT	R	0x0	<p><b>Rx Data Buffer Threshold Status</b></p> <p>The Host Controller shall set this bit to 1'b1 when the number of entries in the Rx Data Queue is <math>\geq</math> the <b>Receive Buffer</b> threshold (i.e., meets or exceeds the value in field <b>RX_BUF_THLD</b> in register <b>DATA_BUFFER_THLD_CTRL</b>).</p> <p>The Host Controller shall automatically clear this field to 1'b0 when the number of Rx Data Queue entries falls below the <b>RX_BUF_THLD</b> threshold.</p> <p>See <b>Section 6.8.1</b> for operational details.</p>
1 [0]	TX_THLD_STAT	R	0x0	<p><b>Tx Data Buffer Threshold Status</b></p> <p>The Host Controller shall set this bit to 1'b1 when the number of available entries in the Tx Data Queue is <math>\geq</math> the <b>Transmit Buffer</b> threshold (i.e., meets or exceeds the value in field <b>TX_BUF_THLD</b> in register <b>DATA_BUFFER_THLD_CTRL</b>).</p> <p>The Host Controller shall automatically clear this field to 1'b0 when the number of available Tx Data Queue entries falls below the <b>TX_BUF_THLD</b> threshold.</p> <p>See <b>Section 6.8.1</b> for operational details.</p>

### 7.5.10 PIO Interrupt Status Enable (PIO\_INTR\_STATUS\_ENABLE) (PIO + 0x24)

The PIO Interrupt Status Enable register enables reporting of outstanding interrupts.

The bit fields in this register act as Interrupt Status Enable Mask bits, according to the interrupt register model described in *Section 6.14*. Writes to these fields shall enable or disable the visibility of an interrupt condition in the Status Register. The value of each field shall determine whether the interrupt condition will be passed to the Interrupt Signal Mask to result in an Interrupt trigger; and will also determine whether software might see the interrupt condition using polling (i.e., without using an interrupt service routine).

**Table 46 PIO Interrupt Status Enable (PIO\_INTR\_STATUS\_ENABLE) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
22 [31:10]	RESERVED	—	—	—
1 [9]	TRANSFER_ERR_STAT_EN	R/W	0x0	<b>Transfer Error Status Enable</b> Enables the corresponding interrupt bit. When set to 1'b1, <b>TRANSFER_ERR_STAT</b> interrupts shall be logged.
3 [8:6]	RESERVED	—	—	—
1 [5]	TRANSFER_ABORT_STAT_EN	R/W	0x0	<b>Transfer Abort Status Enable</b> Enables the corresponding interrupt bit. When set to 1'b1, <b>TRANSFER_ABORT_STAT</b> interrupts shall be logged.
1 [4]	RESP_READY_STAT_EN	R/W	0x0	<b>Response Ready Status Enable</b> Enables the corresponding interrupt bit. When set to 1'b1, <b>RESP_READY_STAT</b> interrupts shall be logged.
1 [3]	CMD_QUEUE_READY_STAT_EN	R/W	0x0	<b>Command Queue Ready Status Enable</b> Enables the corresponding interrupt bit. When set to 1'b1, <b>CMD_QUEUE_READY_STAT</b> interrupts shall be logged.
1 [2]	IBI_STATUS_THLD_STAT_EN	R/W	0x0	<b>IBI Status Threshold Status Enable</b> Enables the corresponding interrupt bit. When set to 1'b1, <b>IBI_STATUS_THLD_STAT</b> interrupts shall be logged.
1 [1]	RX_THLD_STAT_EN	R/W	0x0	<b>Rx Data Buffer Threshold Status Enable</b> Enables the corresponding interrupt bit. When set to 1'b1, <b>RX_THLD_STAT</b> interrupts shall be logged.

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [0]	TX_THLD_STAT_EN	R/W	0x0	<b>Tx Data Buffer Threshold Status Enable</b> Enables the corresponding interrupt bit. When set to 1'b1, TX_THLD_STAT interrupts shall be logged.

### 7.5.11 PIO Interrupt Signal Enable (PIO\_INTR\_SIGNAL\_ENABLE) (PIO + 0x28)

The PIO Interrupt Signal Enable register enables signaling of outstanding interrupts received by the Host Controller.

The fields in this register act as Interrupt Signal Enable Mask bits, according to the interrupt register model described in *Section 6.14*. Writes to these fields shall enable or disable the delivery of an interrupt condition to the Host (i.e., the Interrupt trigger).

**Table 47 PIO Interrupt Signal Enable (PIO\_INTR\_SIGNAL\_ENABLE) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
22 [31:10]	RESERVED	—	—	—
1 [9]	TRANSFER_ERR_SIGNAL_EN	R/W	0x0	<b>Transfer Error Signal Enable</b> When set to 1'b1 and field <b>TRANSFER_ERR_STAT</b> is set, The Host Controller asserts an interrupt to the Host.
3 [8:6]	RESERVED	—	—	—
1 [5]	TRANSFER_ABORT_SIGNAL_EN	R/W	0x0	<b>Transfer Abort Signal Enable</b> When set to 1'b1 and field <b>TRANSFER_ABORT_STAT</b> is set, the Host Controller asserts an interrupt to the Host.
1 [4]	RESP_READY_SIGNAL_EN	R/W	0x0	<b>Response Ready Signal Enable</b> When set to 1'b1 and field <b>RESP_READY_STAT</b> is set, the Host Controller asserts an interrupt to the Host.
1 [3]	CMD_QUEUE_READY_SIGNAL_EN	R/W	0x0	<b>Command Queue Ready Signal Enable</b> When set to 1'b1 and field <b>CMD_QUEUE_READY_STAT</b> is set, the Host Controller asserts an interrupt to the Host.
1 [2]	IBI_STATUS_THLD_SIGNAL_EN	R/W	0x0	<b>IBI Status Threshold Signal Enable</b> When set to 1'b1 and field <b>IBI_STATUS_THLD_STAT</b> is set, the Host Controller asserts an interrupt to the Host.

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [1]	RX_THLD_SIGNAL_EN	R/W	0x0	<b>Rx Data Buffer Threshold Signal Enable</b> When set to 1'b1 and field <b>RX_THLD_STAT</b> is set, the Host Controller asserts an interrupt to the Host.
1 [0]	TX_THLD_SIGNAL_EN	R/W	0x0	<b>Tx Data Buffer Threshold Signal Enable</b> When set to 1'b1 and field <b>TX_THLD_STAT</b> is set, the Host Controller asserts an interrupt to the Host.

### 7.5.12 PIO Interrupt Force (PIO\_INTR\_FORCE) (PIO + 0x2C)

The PIO Interrupt Force register is used to force a specific interrupt. It can be used for debugging purposes.

Writes to fields in this register act as Interrupt-force programming inputs, according to the interrupt register model described in *Section 6.14*.

**Table 48** PIO Interrupt Force (PIO\_INTR\_FORCE) Register

Size [Bits]	Field Name	Memory Access	Reset Value	Description
<b>22</b> [31:10]	RESERVED	—	—	—
<b>1</b> [9]	TRANSFER_ERR_FORCE	W	0x0	For software testing, when set to 1'b1, forces the corresponding interrupt to be sent to the Host, if the corresponding fields are set in registers <b>PIO_INTR_STATUS_ENABLE</b> and <b>PIO_INTR_SIGNAL_ENABLE</b> (see <i>Section 7.5.10</i> and <i>Section 7.5.11</i> ).
<b>3</b> [8:6]	RESERVED	—	—	—
<b>1</b> [5]	TRANSFER_ABORT_FORCE	W	0x0	For software testing, when set to 1'b1, forces the corresponding interrupt to be sent to the Host, if the corresponding fields are set in registers <b>PIO_INTR_STATUS_ENABLE</b> and <b>PIO_INTR_SIGNAL_ENABLE</b> (see <i>Section 7.5.10</i> and <i>Section 7.5.11</i> ).
<b>1</b> [4]	RESP_READY_FORCE	W	0x0	For software testing, when set to 1'b1, forces the corresponding interrupt to be sent to the Host, if the corresponding fields are set in registers <b>PIO_INTR_STATUS_ENABLE</b> and <b>PIO_INTR_SIGNAL_ENABLE</b> (see <i>Section 7.5.10</i> and <i>Section 7.5.11</i> ).
<b>1</b> [3]	CMD_QUEUE_READY_FORCE	W	0x0	
<b>1</b> [2]	IBI_THLD_FORCE	W	0x0	
<b>1</b> [1]	RX_THLD_FORCE	W	0x0	
<b>1</b> [0]	TX_THLD_FORCE	W	0x0	

### 7.5.13 PIO Control (PIO\_CONTROL) (PIO + 0x30)

The PIO Control register is used to control overall operation and status of the PIO Queues. It can be used to enable/disable access to the PIO Queues, control the execution of enqueued Command Descriptors, or abort (i.e., forcibly stop) a current Transfer Command.

**Note:**

*The values written to this register and to register **HC\_CONTROL** (Section 7.4.2) shall control overall Host Controller operation if PIO Mode is selected. Refer to **Figure 11** in Section 6.5.1 for more details on the PIO initialization process.*

**Table 49 PIO Control (PIO\_CONTROL) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
<b>29</b> [31:3]	RESERVED	—	—	—
<b>1</b> [2]	ABORT	R/W	0x0	<p><b>PIO Abort Request</b>            Allows software to forcibly stop execution of the current Command Descriptor, and hold any remaining enqueued Command Descriptors. See <b>Section 6.5.6</b> for operational details.            Software shall only toggle this bit if field <b>ENABLE</b> has a value of 1'b1 (PIO Queues Enabled).</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b1: PIO Queue Abort request</li> <li>• 1'b0: PIO Queue Abort clear request (i.e., resume processing)</li> </ul>
<b>1</b> [1]	RS	R/W	0x0	<p><b>PIO Run / Stop Request</b>            Allows software to start or stop execution of enqueued Command Descriptors.            Software may use this bit to dynamically pause operation of the Command Queue, and hold any remaining enqueued Command Descriptors. The Host Controller shall stop execution gracefully at the boundary after the currently executing Command Descriptor.            Software shall only toggle this bit if field <b>ENABLE</b> has a value of 1'b1 (PIO Queues Enabled).</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b1: PIO Queue Start request</li> <li>• 1'b0: PIO Queue Stop request</li> </ul>
<b>1</b> [0]	ENABLE	R/W	0x1	<p><b>PIO Queues Enable Request</b>            Allows software to enable or disable access to all PIO Queues, if PIO Mode is selected (see register <b>HC_CONTROL</b>).            While access is disabled, software may not read from or write to any PIO Queue registers.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b1: PIO Queues Enable request</li> <li>• 1'b0: PIO Queues Disable request</li> </ul>

## 7.6 Ring Headers Specific Registers

The Host Controller has two transfer Modes: PIO Mode, and DMA Mode for transfers. In DMA Mode, the Host Controller exposes one or more Ring Headers which the Driver uses to allocate Ring-related memory. Rings are defined in Bundles of Command/Response Rings, and Event Rings.

The number of Ring Bundles supported is defined by Host Controller, and may be limited by the Driver, with a maximum of eight Ring Bundles in total.

All Ring-related registers appear in the Ring Headers section of the Register Map, beginning with register Ring Headers Preamble which defines how many Ring Bundles the Host Controller supports, and the size of the Ring Header Descriptor structure (expressed as a power of 2). The Preamble also allows the Driver to limit the number of Rings in use. This is followed by the Ring Header Offsets.

The Ring Headers Preamble and Offsets registers are defined as relative offsets, starting from the location indicated by register **RING\_HEADERS\_SECTION\_OFFSET** (see *Section 7.4.13*), if DMA Mode is supported.

### 7.6.1 Ring Headers Preamble (RHS\_CONTROL) (RHS + 0x0)

The first register in the Ring Headers section is a Preamble register controlling all Rings. It states how many Rings the Host Controller hardware supports, and how many Rings are enabled by the Driver.

**Table 50 Ring Headers Preamble (RHS\_CONTROL) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
8 [31:24]	PREAMBLE_SIZE	R	0x10	<p><b>Preamble Section Size</b> Length in DWORDs of this Preamble Section.</p> <p><b>Note:</b> <i>The default value of 0x10 allows for 8 Ring Headers, and is aligned to a power of 2.</i></p>
8 [23:16]	HEADER_SIZE	R	0x5	<p><b>Ring Header Size</b> Size in DWORDs per Ring Header Descriptor structure, expressed as a power of 2. I.e., structure size is <math>(1 \ll \text{HEADER\_SIZE})</math> DWORDs.</p> <p><b>Example:</b> A HEADER_SIZE value of 3 gives a Ring Header Descriptor size of <math>2^3</math> DWORDs = 8 DWORDs = 32 bytes.</p> <p><b>Reset Value:</b> 0x5, giving a default Ring Header Descriptor size of 0x80 bytes = 128 bytes = 32 DWORDs.</p>
8 [15:8]	RESERVED	-	-	-
4 [7:4]	MAX_HEADER_COUNT_CAPABILITY	R	0x4	<p><b>HC Maximum Header Count</b> Maximum number of Ring Bundles supported by the Host Controller hardware.</p> <p><b>Valid Values:</b> 0-8</p>
4 [3:0]	MAX_HEADER_COUNT	R/W	0x0	<p><b>Driver Maximum Header Count</b> Maximum number of Ring Headers currently enabled by the software.</p> <p>When changing this value, all Ring Bundles (Ring Headers) must be in Disabled state.</p> <p><b>Note:</b> <i>The software may enable fewer Ring Bundles than the MAX_HEADER_COUNT_CAPABILITY hardware maximum.</i></p> <p><b>Valid Values:</b> 0-8</p>

### 7.6.2 Ring Header 0 Offset (RH0\_OFFSET) (RHS + 0x4)

5903 This register holds the Memory Map offset to the first Ring Header Descriptor.

5904 **Table 51 Ring Header Descriptor Offset (RH0\_OFFSET) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	OFFSET	R	-	<b>Ring Header 0 Descriptor Offset</b> HCI Base Address relative offset to Ring Header Descriptor for Ring Bundle index 0.

### 7.6.3 Ring Header 1 Offset (RH1\_OFFSET) (RHS +0x8)

5905 This register holds the Memory Map offset to the second Ring Header Descriptor.

5906 **Table 52 Ring Header Descriptor Offset (RH1\_OFFSET) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	OFFSET	R	-	<b>Ring Header 1 Descriptor Offset</b> HCI Base Address relative offset to Ring Header Descriptor for Ring Bundle index 1.

### 7.6.4 Ring Header 2 Offset (RH2\_OFFSET) (RHS +0xC)

5907 This register holds the Memory Map offset to the third Ring Header Descriptor.

5908 **Table 53 Ring Header Descriptor Offset (RH2\_OFFSET) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	OFFSET	R	-	<b>Ring Header 2 Descriptor Offset</b> HCI Base Address relative offset to Ring Header Descriptor for Ring Bundle index 2.

### 7.6.5 Ring Header 3 Offset (RH3\_OFFSET) (RHS + 0x10)

5909 This register holds the Memory Map offset to the fourth Ring Header Descriptor.

5910 **Table 54 Ring Header Descriptor Offset (RH3\_OFFSET) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	OFFSET	R	-	<b>Ring Header 3 Descriptor Offset</b> HCI Base Address relative offset to Ring Header Descriptor for Ring Bundle index 3.

### 7.6.6 Ring Header 4 Offset (RH4\_OFFSET) (RHS + 0x14)

5911 This register holds the Memory Map offset to the fifth Ring Header Descriptor.

5912 **Table 55 Ring Header Descriptor Offset (RH4\_OFFSET) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	OFFSET	R	-	<b>Ring Header 4 Descriptor Offset</b> HCI Base Address relative offset to Ring Header Descriptor for Ring Bundle index 4.

### 7.6.7 Ring Header 5 Offset (RH5\_OFFSET) (RHS + 0x18)

5913 This register holds the Memory Map offset to the sixth Ring Header Descriptor.

5914 **Table 56 Ring Header Descriptor Offset (RH5\_OFFSET) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	OFFSET	R	-	<b>Ring Header 5 Descriptor Offset</b> HCI Base Address relative offset to Ring Header Descriptor for Ring Bundle index 5.

### 7.6.8 Ring Header 6 Offset (RH6\_OFFSET) (RHS + 0x1C)

5915 This register holds the Memory Map offset to the seventh Ring Header Descriptor.

5916 **Table 57 Ring Header Descriptor Offset (RH6\_OFFSET) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	OFFSET	R	-	<b>Ring Header 6 Descriptor Offset</b> HCI Base Address relative offset to Ring Header Descriptor for Ring Bundle index 6.

### 7.6.9 Ring Header 7 Offset (RH7\_OFFSET) (RHS + 0x20)

5917 This register holds the Memory Map offset to the eighth Ring Header Descriptor.

5918 **Table 58 Ring Header Descriptor Offset (RH7\_OFFSET) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	OFFSET	R	-	<b>Ring Header 7 Descriptor Offset</b> HCI Base Address relative offset to Ring Header Descriptor for Ring Bundle index 7.

### 7.6.10 Ring Header Descriptor (RH +)

The following registers comprise one Ring Header Descriptor, and are defined for each Ring Bundle that the Host Controller supports. The starting offset of each Ring Header Descriptor is linked from registers **RH0\_OFFSET** through **RH7\_OFFSET** (see *Section 7.6.2* through *Section 7.6.9*).

Relative offsets of these registers are shown accordingly (i.e., from the starting offset of the corresponding Ring Header).

The layout of registers within a Ring Header Descriptor is shown in *Section 6.6.4*.

#### 7.6.10.1 CR Setup (CR\_SETUP) (RH + 0x0)

The Command/Response Ring Setup register provides the sizes of the Transfer Descriptor structures in the Command Ring, and the corresponding Response Descriptor structures in the Response Ring. This register also allows the Driver to set the overall size (i.e., number of entries) for both of these Rings.

The Driver shall read fields **XFER\_STRUCT\_SIZE** and **RESP\_STRUCT\_SIZE**, and allocate physical memory buffers with sizes (in bytes) that are at least **XFER\_STRUCT\_SIZE \* RING\_SIZE** (for the Command Ring) and **RESP\_STRUCT\_SIZE \* RING\_SIZE** (for the Response Ring).

- If either Ring does not use Scatter-Gather, or if Scatter-Gather is not supported (per *Section 6.2.1*), then the single memory buffer for the Ring shall be contiguous, and shall have a size sufficient to hold all structures in the Ring.

If Scatter-Gather is supported and enabled for either Ring (per *Section 6.2.1*), then the Driver shall allocate an array of Memory Descriptor structures (see *Section 8.7*), and each shall be configured to point to an allocated memory buffer for that portion of the Ring. Note that each portion shall be a multiple of the Ring's structure size, and the sum of these buffers (i.e., all the portions of the Ring) described by all such Memory Descriptors shall be sufficient to hold all structures in the Ring (i.e., the value of **RING\_SIZE**, which applies to both Command and Response Rings).

The Driver shall update field **RING\_SIZE** before starting the Ring Bundle. See *Section 6.6.1* for more details on Ring Bundle initialization.

If the Driver sets **RING\_SIZE** to 0, then the Host Controller shall consider the Command/Response Rings to be not allocated, and shall disable Command/Response processing for this Ring Bundle.

5944

**Table 59 Command/Response Ring Control (CR\_SETUP) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
8 [31:24]	XFER_STRUCT_SIZE	R	IMPL	<p><b>Command/Transfer Structure Size</b>            Transfer Descriptor structure size (in bytes).            Note that the Transfer Descriptor structure includes one Command Descriptor (see <b>Section 8.3</b>).            The minimum size shall be 20 bytes. An implementer may optionally define a larger value, either for optimized alignment or for vendor-specific extensions (see note below table).</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>0 – 19:</b> Illegal values, do not use</li> <li>• <b>20 (minimum value):</b> A single Transfer Descriptor with no additional bytes</li> <li>• <b>All other legal values</b> (i.e., multiples of 4): Available for vendor-specific extensions</li> </ul>
8 [23:16]	RESP_STRUCT_SIZE	R	IMPL	<p><b>Response Structure Size</b>            Response Descriptor structure size (in bytes).            The minimum size shall be 4 bytes. An implementer may optionally define a larger value, either for optimized alignment or for vendor-specific extensions (see note below table).</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>0 – 3:</b> Illegal values, do not use</li> <li>• <b>4 (minimum value):</b> A single Response Descriptor with no additional bytes</li> <li>• <b>All other legal values</b> (i.e., multiples of 4): Available for vendor-specific extensions</li> </ul>
8 [15:8]	RESERVED	—	—	—
8 [7:0]	RING_SIZE	R/W	0x0	<p><b>Command/Response Ring Size</b>            Command/Response Ring size, expressed as a number of paired Transfer Descriptors and Response Descriptors.            The Driver software shall allocate <math>(XFER\_STRUCT\_SIZE * RING\_SIZE)</math> bytes for the Command Ring, and <math>(RESP\_STRUCT\_SIZE * RING\_SIZE)</math> bytes for the Response Ring.</p> <p><b>Valid Values:</b></p> <ul style="list-style-type: none"> <li>• <b>0:</b> Valid if the Command/Response Rings are disabled for this Ring Bundle (i.e., only IBI Rings might be enabled)</li> <li>• <b>1:</b> Illegal value, do not use</li> <li>• <b>2 – 255:</b> Valid if the Command and Response Rings are enabled for this Ring Bundle</li> </ul>

5945

**Note:**

5946

If fields **XFER\_STRUCT\_SIZE** and/or **RESP\_STRUCT\_SIZE** are larger than the minimum structure sizes (i.e., 20 bytes or 4 bytes, respectively) then the implementer has defined a larger structure size for a specific purpose, and the Driver must allocate sufficient memory in the Ring to accomodate the larger structure size. In this case, the use of such additional bytes in each structure shall be defined by the implementer, and the Driver must not assume that these additional bytes are available for general use. If larger structure sizes are used, then implementers should provide sizes that are DWORD-aligned, for optimal system performance.

5947

5948

5949

5950

5951

5952

5953

A value of 0 in field **RING\_SIZE** effectively disables the Command/Response Ring processing for a particular Ring Bundle, and allows the Driver to enable processing for only the IBI Rings (i.e., in register **IBI\_SETUP**) for this Ring Bundle. If the Command/Response Rings are enabled, then the minimum Ring size shall be 2 entries, although the smallest usable size might be somewhat greater (i.e., at least 4 entries) per the use case, since the size of a Ring is primarily limited by available Host system memory.

5954

5955

5956

5957

5958

### 7.6.10.2 IBI Setup (IBI\_SETUP) (RH + 0x4)

The IBI Ring Setup register provides the sizes of the IBI Status Descriptor structures and the IBI Data Chunk, and the overall size of the IBI Ring.

The Driver shall read the **IBI\_STATUS\_STRUCT\_SIZE** field and allocate physical memory buffer(s) of at least (**IBI\_STATUS\_STRUCT\_SIZE** \* **IBI\_STATUS\_RING\_SIZE**) bytes for the IBI Status Ring.

Similarly, the Driver shall also allocate physical memory buffer(s) for the IBI Data Ring, based on the configured Data Chunk Size (i.e., field **CHUNK\_SIZE**) and the number of Data Chunks (i.e., field **CHUNK\_COUNT**).

- If the either Ring does not use Scatter-Gather, or if Scatter-Gather is not supported per *Section 6.2.1*, then the single memory buffer for the Ring shall be contiguous, and shall have a size sufficient to hold all structures/chunks in the Ring.
- If Scatter-Gather is supported and enabled for either Ring, per *Section 6.2.1*, then the Driver shall allocate an array of Memory Descriptor structures (see *Section 8.7*), and each shall be configured to point to an allocated memory buffer for that portion of the Ring. Note that each portion shall be a multiple of the Ring's structure/chunk size, and the sum of these buffers (i.e., all the portions of the Ring) described by all such Memory Descriptors shall be sufficient to hold all structures/chunks in the Ring:
  - For the IBI Status Ring, this shall be determined by the value of field **IBI\_STATUS\_RING\_SIZE**.
  - For the IBI Data Ring, this shall be determined by the value of field **CHUNK\_COUNT**.

The Driver shall update fields **IBI\_STATUS\_RING\_SIZE** and **CHUNK\_COUNT** before starting the Ring Bundle. See *Section 6.6.1* for more details on Ring Bundle initialization.

If the Driver sets **IBI\_STATUS\_RING\_SIZE** and **CHUNK\_COUNT** to 0, then the Host Controller shall consider the IBI Rings to be not allocated, and shall not serve any IBIs for this Ring Bundle.

5980

**Table 60 IBI Ring Control (IBI\_SETUP) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
8 [31:24]	IBI_STATUS_STRUCT_SIZE	R	0x4	<b>IBI Status Structure Size</b> IBI Status Descriptor structure size (in bytes).
8 [23:16]	IBI_STATUS_RING_SIZE	R/W	0x0	<b>IBI Status Ring Size</b> IBI Status Ring size (in IBI Status Descriptor entries). <b>Valid Values:</b> <ul style="list-style-type: none"> <li>• <b>0:</b> Valid if the IBI Rings are disabled for this Ring Bundle</li> <li>• <b>1:</b> Illegal value, do not use</li> <li>• <b>2 – 255:</b> Valid if the IBI Rings are enabled for this Ring Bundle</li> </ul>
3 [15:13]	RESERVED	—	—	—
3 [12:10]	CHUNK_SIZE	R/W	0x0	<b>IBI Data Ring Data Chunk Size</b> Size per Data Chunk on the IBI Data Ring. <b>Values:</b> <ul style="list-style-type: none"> <li>• <b>0:</b> 4 bytes per Data Chunk</li> <li>• <b>1:</b> 8 bytes per Data Chunk</li> <li>• <b>2:</b> 16 bytes per Data Chunk</li> <li>• <b>3:</b> 32 bytes per Data Chunk</li> <li>• <b>4:</b> 64 bytes per Data Chunk</li> <li>• <b>5:</b> 128 bytes per Data Chunk</li> <li>• <b>6:</b> 256 bytes per Data Chunk</li> <li>• <b>7:</b> Reserved, do not use</li> </ul>
10 [9:0]	CHUNK_COUNT	R/W	0x0	<b>IBI Data Ring Chunk Count</b> Number of Data Chunks on IBI Data Ring. The size in bytes of the IBI Data Ring is <b>CHUNK_COUNT * CHUNK_SIZE</b> . <b>Valid Values:</b> 0-0x3FF

5981

**Note:**

5982  
5983  
5984  
5985

A value of 0 in field **IBI\_STATUS\_RING\_SIZE** effectively disables the IBI Ring processing for a particular Ring Bundle. If the IBI Rings are enabled, then the minimum Ring size shall be 2 entries, although the smallest usable size might be somewhat greater (i.e., at least 4 entries) per the use case, since the size of a Ring is primarily limited by available Host system memory.

### 7.6.10.3 Chunk Control (CHUNK\_CONTROL) (RH + 0x8)

The Chunk Control register is a monotonic counter that facilitates shared control of Chunks between the Driver and the Host Controller hardware.

5986

**Table 61 Chunk Control (CHUNK\_CONTROL) Register**

5987

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	CHUNK_COUNTER	R/W	0x0	<p><b>Chunk Counter</b></p> <p>This is a monotonic counter that the Driver would increment by the number of Chunks freed up, providing the Host Controller with the number of Data Chunks freed (see <a href="#">Section 6.6.3.2</a>).</p> <p>The Driver shall reset this field before transitioning the Ring Bundle to the Enabled state.</p>

#### 7.6.10.4 Ring Interrupt Status (RH\_INTR\_STATUS) (RH + 0x10)

The Interrupt Status register reflects the status of interrupts received by the Host Controller, for a specific Ring Bundle instance.

The status fields are R/W1C (write 1 to clear).

The bit fields in this register act as Status Bits, according to the interrupt register model described in [Section 6.14](#). These bits shall be set (i.e., the “Status bit set” action will occur) if the interrupt condition is detected, and if the corresponding bit in the Interrupt Status Enable Mask is set to 1'b1.

**Table 62 Ring Interrupt Status (RH\_INTR\_STATUS) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
19 [31:13]	RESERVED	—	—	—
1 [12]	IBI_READY_STAT	R/W1C	0x0	<b>IBI Ready Status</b> The Host Controller shall write 1'b1 to this field to indicate that an IBI has been received.
1 [11]	TRANSFER_COMPLETION_STAT	R/W1C	0x0	<b>Transfer Completion Status</b> The Host Controller shall write 1'b1 to this field to indicate successful completion of a transfer, for all Transfer Descriptors featuring IOC.
1 [10]	RING_OP_STAT	R/W1C	0x0	<b>Ring Operation Status</b> The Host Controller shall write 1'b1 to this field to indicate that the Ring's state has changed to <b>RUNNING</b> , <b>ABORTED</b> , or <b>STOPPED</b> (see <a href="#">Section 6.6.2</a> and <a href="#">Section 6.6.6</a> ).
1 [9]	TRANSFER_ERR_STAT	R/W1C	0x0	<b>Transfer Error Status</b> The Host Controller shall write 1'b1 to this field to indicate that an error (i.e., any error) has occurred during an I3C Bus transfer. The Error Type for this error event can be found in the Response Descriptor structure corresponding to this transfer.
2 [8:7]	RESERVED	—	—	—
1 [6]	IBI_RING_FULL_STAT	R/W1C	0x0	<b>IBI Ring Full Condition Error Status</b> The Host Controller shall write 1'b1 to this field to indicate lack of space on IBI Status or IBI Data Rings upon receiving IBI.

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [5]	TRANSFER_ABORT_STAT	R/W1C	0x0	<b>Transfer Abort Status</b> The Host Controller shall write 1'b1 to this field to indicate that a transfer operation has been aborted.
5 [4:0]	RESERVED	-	-	-

#### 7.6.10.5 Ring Interrupt Status Enable (RH\_INTR\_STATUS\_ENABLE) (RH + 0x14)

The Ring Interrupt Status Enable register controls reporting of outstanding interrupts, for specific Ring Bundle instances.

The fields in this register act as Interrupt Status Enable Mask bits, according to the interrupt register model described in *Section 6.14*. Writes to these fields shall enable or disable the visibility of an interrupt condition in the Status Register. The value of each field shall determine whether the interrupt condition will be passed to the Interrupt Signal Mask to result in an Interrupt trigger; and will also determine whether software sees the interrupt condition using polling (i.e., without using an interrupt service routine).

**Table 63 Ring Interrupt Status Enable (RH\_INTR\_STATUS\_ENABLE) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
<b>19</b> [31:13]	RESERVED	—	—	—
<b>1</b> [12]	<b>IBI_READY_STAT_EN</b>	R/W	0x0	<b>IBI Ready Status Enable</b> Enables the corresponding interrupt bit. When set to 1'b1, <b>IBI_READY_STAT</b> interrupts shall be logged.
<b>1</b> [11]	<b>TRANSFER_COMPLETION_STAT_EN</b>	R/W	0x0	<b>Transfer Completion Status Enable</b> Enables the corresponding interrupt bit. When set to 1'b1, <b>TRANSFER_COMPLETION_STAT</b> interrupts shall be logged.
<b>1</b> [10]	<b>RING_OP_STAT_EN</b>	R/W	0x0	<b>Ring Operation Status Enable</b> Enables the corresponding interrupt bit. When set to 1'b1, <b>RING_OP_STAT</b> interrupts shall be logged.
<b>1</b> [9]	<b>TRANSFER_ERR_STAT_EN</b>	R/W	0x0	<b>Transfer Error Status Enable</b> Enables the corresponding interrupt bit. When set to 1'b1, <b>TRANSFER_ERR_STAT</b> interrupts shall be logged.
<b>2</b> [8:7]	RESERVED	—	—	—
<b>1</b> [6]	<b>IBI_RING_FULL_STAT_EN</b>	R/W	0x0	<b>IBI Ring Full Condition Status Enable</b> Enables the corresponding interrupt bit. When set to 1'b1, <b>IBI_RING_FULL_STAT</b> interrupts shall be logged.

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [5]	TRANSFER_ABORT_STAT_EN	R/W	0x0	<b>Transfer Abort Status Enable</b> Enables the corresponding interrupt bit. When set to 1'b1, <b>TRANSFER_ABORT_STAT</b> interrupts shall be logged.
5 [4:0]	RESERVED	-	-	-

#### 7.6.10.6 Ring Interrupt Signal Enable (RH\_INTR\_SIGNAL\_ENABLE) (RH + 0x18)

The Ring Interrupt Signal Enable register controls signaling of outstanding interrupts received by the Host Controller, for specific Ring Bundle instances.

The fields in this register act as Interrupt Signal Enable Mask bits, according to the interrupt register model described in [Section 6.14](#). Writes to these fields may enable or disable the delivery of an interrupt condition to the Host (i.e., the Interrupt trigger).

**Table 64 Ring Interrupt Signal Enable (RH\_INTR\_SIGNAL\_ENABLE) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
19 [31:13]	RESERVED	-	-	-
1 [12]	IBI_READY_SIGNAL_EN	R/W	0x0	When set to 1'b1, and the corresponding interrupt status field is set in register <a href="#">RH_INTR_STATUS</a> within this same Ring Header (see <a href="#">Section 7.6.10.4</a> ), the Host Controller asserts an interrupt to the Host.
1 [11]	TRANSFER_COMPLETION_SIGNAL_EN	R/W	0x0	When set to 1'b1, and the corresponding interrupt status field is set in register <a href="#">RH_INTR_STATUS</a> within this same Ring Header (see <a href="#">Section 7.6.10.4</a> ), the Host Controller asserts an interrupt to the Host.
1 [10]	RING_OP_SIGNAL_EN	R/W	0x0	
1 [9]	TRANSFER_ERR_SIGNAL_EN	R/W	0x0	When set to 1'b1, and the corresponding interrupt status field is set in register <a href="#">RH_INTR_STATUS</a> within this same Ring Header (see <a href="#">Section 7.6.10.4</a> ), the Host Controller asserts an interrupt to the Host.
2 [8:7]	RESERVED	-	-	
1 [6]	IBI_RING_FULL_SIGNAL_EN	R/W	0x0	When set to 1'b1, and the corresponding interrupt status field is set in register <a href="#">RH_INTR_STATUS</a> within this same Ring Header (see <a href="#">Section 7.6.10.4</a> ), the Host Controller asserts an interrupt to the Host.
1 [5]	TRANSFER_ABORT_SIGNAL_EN	R/W	0x0	
5 [4:0]	RESERVED	-	-	-

### 7.6.10.7 Ring Interrupt Force (RH\_INTR\_FORCE) (RH + 0x1C)

The Ring Interrupt Force register is used to force specific interrupts, for specific Ring Bundle instances. This register can be used for debug purposes.

Writes to fields in this register act as Interrupt-force programming inputs, according to the interrupt register model described in [Section 6.14](#).

**Table 65 Ring Interrupt Force (RH\_INTR\_FORCE) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
<b>19</b> [31:13]	RESERVED	—	—	—
<b>1</b> [12]	<b>IBI_READY_FORCE</b>	W	0x0	For software testing, when set to 1'b1, forces the corresponding interrupt to be sent to the Host, if the corresponding fields are set in registers <b>RH_INTR_STATUS_ENABLE</b> and <b>RH_INTR_SIGNAL_ENABLE</b> within this same Ring Header (see <a href="#">Section 7.6.10.5</a> and <a href="#">Section 7.6.10.6</a> ).
<b>1</b> [11]	<b>TRANSFER_COMPLETION_FORCE</b>	W	0x0	
<b>1</b> [10]	<b>RING_OP_FORCE</b>	W	0x0	
<b>1</b> [9]	<b>TRANSFER_ERR_FORCE</b>	W	0x0	
<b>2</b> [8:7]	RESERVED	—	—	—
<b>1</b> [6]	<b>IBI_RING_FULL_FORCE</b>	W	0x0	For software testing, when set to 1'b1, forces the corresponding interrupt to be sent to Host, if the corresponding fields are set in registers <b>RH_INTR_STATUS_ENABLE</b> and <b>RH_INTR_SIGNAL_ENABLE</b> within this same Ring Header (see <a href="#">Section 7.6.10.5</a> and <a href="#">Section 7.6.10.6</a> ).
<b>1</b> [5]	<b>TRANSFER_ABORT_FORCE</b>	W	0x0	
<b>5</b> [4:0]	RESERVED	—	—	—

### 7.6.10.8 Ring Status (RH\_STATUS) (RH + 0x20)

6015 The Ring Status register allows Ring status to be read.

6016 **Table 66 Ring Control Status (RH\_STATUS) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
<b>28</b> [31:4]	RESERVED	—	—	—
<b>1</b> [3]	LOCKED	R	0x0	<b>Ring Locked</b> Indicates lock status of Ring <b>Values:</b> <ul style="list-style-type: none"> <li>• 1'b1: Ring locked</li> <li>• 1'b0: Ring not locked</li> </ul>
<b>1</b> [2]	ABORTED	R	0x0	<b>Ring Aborted</b> <b>Values:</b> <ul style="list-style-type: none"> <li>• 1'b1: Ring processing aborted per <b>Section 6.6.6</b>. If the Ring is aborted, then it shall wait for Ring Abort condition clearing and Enqueue Pointer Write (Doorbell) in order to resume operation.</li> <li>• 1'b0: Ring not aborted, or prior Ring Abort was cleared (i.e., to resume operation)</li> </ul>
<b>1</b> [1]	RUNNING	R	0x0	<b>Ring Running</b> <b>Values:</b> <ul style="list-style-type: none"> <li>• 1'b1: Ring running</li> <li>• 1'b0: Ring not running (stopped) If the Ring is not running, then the Enqueue Pointer update shall not trigger the processing of Transfer Commands for the Command/Response Ring Pair.</li> </ul>
<b>1</b> [0]	ENABLED	R	0x0	<b>Ring Enabled</b> <b>Values:</b> <ul style="list-style-type: none"> <li>• 1'b1: Ring enabled</li> <li>• 1'b0: Ring disabled If the Ring is disabled, then no operation on the Ring shall be possible.</li> </ul>

### 7.6.10.9 Ring Control (RH\_CONTROL) (RH + 0x24)

6017

The Ring Control register allows Ring status changes to be requested.

6018

**Table 67 Ring Control (RH\_CONTROL) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
<b>29</b> [31:3]	RESERVED	—	—	—
<b>1</b> [2]	<b>ABORT</b>	R/W	0x0	<p><b>Ring Abort Request</b>            Allows Driver to abort processing of current transfer (at earliest opportunity) per <b>Section 6.6.6</b>. In the <b>ABORTED</b> state, the Ring shall block processing of any subsequent enqueued Transfer Descriptors  <b>Values:</b>            1'b1: Ring Abort operation request            1'b0: Ring Abort operation clear (i.e., resume processing)</p>
<b>1</b> [1]	<b>RS</b>	R/W	0x0	<p><b>Ring Run / Stop Request</b>            If the Ring Bundle is running, then the <b>RUNNING</b> bit in register <b>RH_STATUS</b> shall have the value 1'b1.            The Driver may use this bit to dynamically pause operation of a given Ring Bundle. The Host Controller shall stop the Ring Bundle gracefully at the first possible opportunity (boundary).            The software Driver shall only toggle this bit if the <b>ENABLE</b> field already has value 1'b1 (Ring Enable requested).  <b>Values:</b>            1'b1: Ring Start request            1'b0: Ring Stop request</p>
<b>1</b> [0]	<b>ENABLE</b>	R/W	0x0	<p><b>Ring Enable Request</b>            If the Ring Bundle is enabled for hardware processing, then the <b>ENABLED</b> bit in register <b>RH_STATUS</b> shall have the value 1'b1.            A transition of this bit to 1'b1 (Enable) shall reset all hardware internal pointers (if any).  <b>Values:</b>            1'b1: Ring Enable request            1'b0: Ring Disable request</p>

#### 7.6.10.10 Ring Operation 1 (RH\_OPERATION1) (RH + 0x28)

6019 Ring Operation 1 Register contains Ring pointers for dequeue and enqueue operations.

6020 **Table 68 Ring Operation 1 (RH\_OPERATION1) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
8 [31:24]	RESERVED	-	-	-
8 [23:16]	IBI_DEQ_PTR	R/W	-	<b>IBI Status Ring Dequeue Pointer</b>
8 [15:8]	CR_SW_DEQ_PTR	R/W	-	<b>Command/Response Ring Software Dequeue Pointer</b>
8 [7:0]	CR_ENQ_PTR	R/W	-	<b>Command/Response Ring Enqueue Pointer</b>

#### 7.6.10.11 Ring Operation 2 (RH\_OPERATION2) (RH + 0x2C)

6021 Ring Operation 2 Register contains Ring pointers for dequeue and enqueue operations.

6022 **Table 69 Ring Operation 2 (RH\_OPERATION2) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
8 [31:24]	RESERVED	-	-	-
8 [23:16]	IBI_ENQ_PTR	R	0x0	<b>IBI Status Ring Hardware Enqueue Pointer</b>
8 [15:8]	RESERVED	-	-	-
8 [7:0]	CR_DEQ_PTR	R	0x0	<b>Command/Response Ring Dequeue Pointer</b>

### 7.6.10.12 Command Ring Base Address Low (RH\_CMD\_RING\_BASE\_LO) (RH + 0x30)

Register Command Ring Base Address Low indicates the location of the Command Ring, in combination with register Command Ring Base Address High.

**Table 70 Command Ring Base Address Low (RH\_CMD\_RING\_BASE\_LO) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	BASE_LO	R/W	0x0	<b>Command Ring Base Address Low</b> Lower 32 bits of pointer to physical memory allocated for the Command Ring. The Command Ring is DWORD aligned, so the last two Address bits will always be 2'b00.

### 7.6.10.13 Command Ring Base Address High (RH\_CMD\_RING\_BASE\_HI) (RH + 0x34)

Register Command Ring Base Address High indicates the location of the Command Ring, in combination with register Command Ring Base Address Low.

**Table 71 Command Ring Base Address High (RH\_CMD\_RING\_BASE\_HI) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	BASE_HI	R/W	0x0	<b>Command Ring Base Address High</b> Upper 32 bits of pointer to physical memory allocated for Command Ring

#### 7.6.10.14 Response Ring Base Address Low (RH\_RESP\_RING\_BASE\_LO) (RH + 0x38)

Register Response Ring Base Address Low indicates the location of the Response Ring, in combination with register Response Ring Base Address High.

Table 72 Response Ring Base Address Low (RH\_RESP\_RING\_BASE\_LO) Register

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	BASE_LO	R/W	0x0	<b>Response Ring Base Address Low</b> Lower 32 bits of pointer to physical memory allocated for the Response Ring. Because the Response Ring is DWORD aligned, the lowest two Address bits will always be 2'b00.

#### 7.6.10.15 Response Ring Base Address High (RH\_RESP\_RING\_BASE\_HI) (RH + 0x3C)

Register Response Ring Base Address High indicates the location of the Response Ring, in combination with register Response Ring Base Address Low.

Table 73 Response Ring Base Address High (RH\_RESP\_RING\_BASE\_HI) Register

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	BASE_HI	R/W	0x0	<b>Response Ring Base Address High</b> Upper 32 bits of pointer to physical memory allocated for Response Ring

#### 7.6.10.16 IBI Status Ring Base Address Low (RH\_IBI\_STATUS\_RING\_BASE\_LO) (RH + 0x40)

6035 Register IBI Status Ring Base Address Low indicates the location of the IBI Status Ring, in combination  
6036 with register IBI Status Ring Base Address High.

6037 **Table 74 IBI Status Ring Base Address Low (RH\_IBI\_STATUS\_RING\_BASE\_LO) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	BASE_LO	R/W	0x0	<b>IBI Status Ring Base Address Low</b> Lower 32 bits of pointer to physical memory allocated for the IBI Status Ring. Because the IBI Status Ring is DWORD aligned, the lowest two Address bits will always be 2'b00.

#### 7.6.10.17 IBI Status Ring Base Address High (RH\_IBI\_STATUS\_RING\_BASE\_HI) (RH + 0x44)

6038 Register IBI Status Ring Base Address High indicates the location of the IBI Status Ring, in combination  
6039 with register IBI Status Ring Base Address Low.

6040 **Table 75 IBI Status Ring Base Address High (RH\_IBI\_STATUS\_RING\_BASE\_HI) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	BASE_HI	R/W	0x0	<b>IBI Status Ring Base Address High</b> Upper 32 bits of pointer to physical memory allocated for IBI Status Ring

#### 7.6.10.18 IBI Data Ring Base Address Low (RH\_IBI\_DATA\_RING\_BASE\_LO) (RH + 0x48)

Register IBI Data Ring Base Address Low indicates the location of the IBI Data Ring, in combination with register IBI Data Ring Base Address High.

**Table 76 IBI Data Ring Base Address Low (RH\_IBI\_DATA\_RING\_BASE\_LO) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	BASE_LO	R/W	0x0	<b>IBI Data Ring Base Address Low</b> Lower 32 bits of pointer to physical memory allocated for the IBI Data Ring. Because the IBI Data Ring is DWORD aligned, the lowest two Address bits will always be 2'b00.

#### 7.6.10.19 IBI Data Ring Base Address High (RH\_IBI\_DATA\_RING\_BASE\_HI) (RH + 0x4C)

Register IBI Data Ring Base Address High indicates the location of the IBI Data Ring, in combination with register IBI Data Ring Base Address Low.

**Table 77 IBI Data Ring Base Address High (RH\_IBI\_DATA\_RING\_BASE\_HI) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	BASE_HI	R/W	0x0	<b>IBI Data Ring Base Address High</b> Upper 32 bits of pointer to physical memory allocated for IBI Data Ring

### 7.6.10.20 Command Ring Scatter-Gather Support (RH\_CMD\_RING\_SG) (RH + 0x50)

The Command Ring Scatter-Gather Support register allows for memory for the Command Ring to be allocated as multiple chunks of physically contiguous memory, using Scatter-Gather allocation per [Section 6.2.1](#).

**Table 78 Command Ring Scatter-Gather Support (RH\_CMD\_RING\_SG) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [31]	BLP	R/W	0x0	<p><b>Buffer vs. List Pointer</b>            Indicates whether the <a href="#">RH_CMD_RING_BASE_HI</a> / <a href="#">RH_CMD_RING_LO</a> pointer (see <a href="#">Section 7.6.10.13</a> and <a href="#">Section 7.6.10.12</a>) points directly to the Buffer, or to a List of Memory Descriptors describing the Buffer.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: BUFFER: Pointer points to physical memory</li> <li>• 1'b1: LIST: Pointer points to physical memory comprising array of Memory Descriptors (i.e., the Ring uses a Scatter-Gather Buffer that might not be physically continuous)</li> </ul>
15 [30:16]	RESERVED	R/W	0x0	–
16 [15:0]	LIST_SIZE	R/W	0x0	<p><b>List Size</b>            List Size in Entries</p>

#### 7.6.10.21 Response Ring Scatter-Gather Support (RH\_RESP\_RING\_SG) (RH + 0x54)

The Response Ring Scatter-Gather Support register allows for memory for Response Ring to be allocated as multiple chunks of physically contiguous memory, using Scatter-Gather allocation (per *Section 6.2.1*).

Table 79 Response Ring Scatter-Gather Support (RH\_RESP\_RING\_SG) Register

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [31]	BLP	R/W	0x0	<b>Buffer vs. List Pointer</b> Indicates whether the <b>RH_RESP_RING_BASE_HI / RH_RESP_RING_LO</b> pointer (see <i>Section 7.6.10.15</i> and <i>Section 7.6.10.14</i> ) points directly to the Buffer, or to a List of Memory Descriptors describing the Buffer. <b>Values:</b> <ul style="list-style-type: none"><li>• <b>1'b0: BUFFER:</b> Pointer points to physical memory</li><li>• <b>1'b1: LIST:</b> Pointer points to physical memory comprising array of Memory Descriptors (i.e., the Ring uses a Scatter-Gather Buffer that might not be physically continuous)</li></ul>
15 [30:16]	RESERVED	R/W	0x0	–
16 [15:0]	LIST_SIZE	R/W	0x0	<b>List Size</b> List Size in Entries

### 7.6.10.22 IBI Status Ring Scatter-Gather (RH\_IBI\_STATUS\_RING\_SG) (RH + 0x58)

The IBI Status Ring Scatter-Gather Support register allows for memory for IBI Status Ring to be allocated as multiple chunks of physically contiguous memory, using Scatter-Gather allocation (per *Section 6.2.1*).

**Table 80 IBI Status Ring Scatter-Gather (RH\_IBI\_STATUS\_RING\_SG) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [31]	BLP	R/W	0x0	<p><b>Buffer vs. List Pointer</b>            Indicates whether the <b>RH_IBI_STATUS_RING_BASE_HI</b> / <b>RH_IBI_STATUS_RING_LO</b> pointer (see <i>Section 7.6.10.17</i> and <i>Section 7.6.10.16</i>) points directly to the Buffer, or to a List of Memory Descriptors describing the Buffer.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>1'b0: BUFFER:</b> Pointer points to physical memory</li> <li>• <b>1'b1: LIST:</b> Pointer points to physical memory comprising array of Memory Descriptors (i.e., the Ring uses a Scatter-Gather Buffer that might not be physically continuous)</li> </ul>
15 [30:16]	RESERVED	R/W	0x0	–
16 [15:0]	LIST_SIZE	R/W	0x0	<p><b>List Size</b>            List Size in Entries</p>

### 7.6.10.23 IBI Data Ring Scatter-Gather Support (RH\_IBI\_DATA\_RING\_SG) (RH + 0x5C)

The IBI Data Ring Scatter-Gather Support register allows for memory for IBI Data Ring to be allocated as multiple chunks of physically contiguous memory, using Scatter-Gather allocation (per *Section 6.2.1*).

**Table 81 IBI Data Ring Scatter-Gather Support (RH\_IBI\_DATA\_RING\_SG) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [31]	BLP	R/W	0x0	<p><b>Buffer Vs. List Pointer</b>            Indicates whether the <b>RH_IBI_DATA_RING_BASE_HI</b> / <b>RH_IBI_DATA_RING_BASE_LO</b> pointer (see <b>Section 7.6.10.19</b> and <b>Section 7.6.10.18</b>) points directly to the Buffer, or to a List of Memory Descriptors describing the Buffer.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: <b>BUFFER</b>: Pointer points to physical memory</li> <li>• 1'b1: <b>LIST</b>: Pointer points to physical memory comprising array of Memory Descriptors (i.e., Scatter-Gather Buffers that are not physically continuous)</li> </ul>
15 [30:16]	RESERVED	R/W	0x0	—
16 [15:0]	LIST_SIZE	R/W	0x0	<p><b>Buffer List Size</b>            List Size in Entries</p>

## 7.7 Extended Capabilities Registers

Extended Capabilities are implemented as a linked list of Extended Capability structures (see [Section 7.3.2](#))

Extended Capabilities registers are defined as relative offsets, starting from the location indicated by register [EXT\\_CAPS\\_SECTION\\_OFFSET](#) (see [Section 7.4.15](#)).

Each Extended Capability structure represents one Extended Capability, as identified by the Extended Capability ID field. [Table 82](#) lists the defined and reserved Extended Capability IDs. Further information for each defined Extended Capability ID appears below in this Section.

**Table 82 Extended Capability IDs**

Extended Capability ID Value	Description	See Section
0x00	Special marker, for end of linked list only	<a href="#">7.3.2</a>
0x01	Hardware Identification	<a href="#">7.7.2</a>
0x02	Controller Extended Configuration	<a href="#">7.7.3</a>
0x03	Multi-Bus Instance <i>(Valid for first Bus Controller instance only)</i>	<a href="#">7.7.4</a>
0x04 – 0x09	Reserved for future definition by MIPI Alliance Software WG	–
0x0A	Target IBI Credit Counters	<a href="#">7.7.5</a>
0x0B	Dead Bus Recovery	<a href="#">7.7.6</a>
0x0C	Debug Specific	<a href="#">7.7.7</a>
0x0D	Scheduled Commands (Handler Type 1, Simple Table)	<a href="#">7.7.8</a>
0x0E	Scheduled Commands (Handler Type 2, Command Sequence)	<a href="#">7.7.9</a>
0x0F	Scheduled Commands (Handler Type 3, Schedule Buffer)	<a href="#">7.7.10</a>
0x10– 0x11	Reserved for future definition by MIPI Alliance Software WG	–
0x12	Standby Controller Mode	<a href="#">7.7.11</a>
0x13 – 0xAF	Reserved for future definition by MIPI Alliance Software WG	–
0xB0 – 0xBF	Target Specific	<a href="#">7.7.12</a>
0xC0 – 0xCF	Vendor Specific <i>(Available for vendor implementations)</i>	<a href="#">7.7.13</a>
0xD0 – 0xFF	Reserved for future definition by MIPI Alliance Software WG	–

**Note:**

Per [Section 7.3.2](#), any Extended Capability IDs that are reserved for future definition might subsequently be defined in a future version of this I3C HCI Specification, or in a special I3C HCI Feature Specification that would extend this I3C HCI Specification to define support for a particular optional feature or capability.

### 7.7.1 Extended Capability Header (EXTCAP\_HEADER)

Every Extended Capability is introduced with a single Extended Capability Header, followed by a number of Capability-specific registers.

**Table 83 Extended Capability Header (EXTCAP\_HEADER) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
8 [31:24]	RESERVED	–	–	–
16 [23:8]	CAP_LENGTH	R	IMPL	<b>Capability Structure Length</b> in DWORDs The length includes the <b>EXTCAP_HEADER</b> , and indicates the base Address of the next Extended Capability in the list.
8 [7:0]	CAP_ID	R	IMPL	<b>Extended Capability ID Values:</b> <ul style="list-style-type: none"><li>• <b>0x00:</b> Not a capability ID, indicates end of the Extended Capability list</li><li>• <b>0x01 – 0xFF:</b> Capability IDs, see <b>Table 82</b> for valid vs. reserved values</li></ul>

**Note:**

The minimum value for field **CAP\_LENGTH** is 1, which indicates an Extended Capability structure with no additional registers beyond the **EXTCAP\_HEADER** register itself. This would not be a valid structure, as it would not have any additional registers.

The base address of the next Extended Capability structure in the linked list can be found by adding the value of field **CAP\_LENGTH** \* 4 to the current Extended Capability structure's base address.

### 7.7.2 Hardware Identification (ID = 0x01)

This Section defines the three available Capability-specific read-only registers under Extended Capability ID 0x01, Hardware Identification: Component Manufacturer, Component Version, and Component Type.

#### 7.7.2.1 Component Manufacturer (COMP\_MANUFACTURER) (+ 0x04)

MIPI-assigned Manufacturer ID for the Host Controller.

**Table 84 Component Manufacturer (COMP\_MANUFACTURER) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	MIPI_VENDOR_ID	R	0x0	<b>MIPI Vendor ID</b> MIPI Manufacturer ID [ <i>MIPI01</i> ].

#### 7.7.2.2 Component Version (COMP\_VERSION) (+ 0x08)

Vendor-assigned component version for the Host Controller.

**Table 85 Component Version (COMP\_VERSION) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	I3C_VER_ID	R	0x0	<b>Host Controller Component Version</b> Vendor-assigned Device version.

#### 7.7.2.3 Component Type (COMP\_TYPE) (+ 0x0C)

Vendor-assigned component type for the Host Controller.

**Table 86 Component Type (COMP\_TYPE) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	I3C_PRODUCT_ID	R	0x0	<b>Host Controller Component Type</b> Vendor-assigned product ID or type.

### 7.7.3 Controller Config (ID = 0x02)

6090 This Section defines the optional Capability-specific read-only register available under Extended Capability  
6091 ID 0x02, Controller Config.

#### 7.7.3.1 Controller Config (CONTROLLER\_CONFIG) (+ 0x04)

6092 Additional optional read-only Controller configuration parameters.

6093 **Table 87 Controller Config (CONTROLLER\_CONFIG) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
26 [31:6]	RESERVED	—	—	—
2 [5:4]	OPERATION_MODE	R	0x1	<b>Controller Operation Mode</b> Host Controller modes supported. <b>Values:</b> <ul style="list-style-type: none"><li>• 0x0: Illegal value, do not use</li><li>• 0x1: Controller Role functionality only (i.e., Active Controller)</li><li>• 0x2: Target Role functionality only (not supported in this version of the I3C HCI Specification)</li><li>• 0x3: Both Controller Role and Target Role functionality (i.e., both Active Controller and Secondary Controller)</li></ul>
4 [3:0]	RESERVED	—	—	—

### 7.7.4 Multi-Bus Instance (ID = 0x03)

This Section defines the two optional Capability-specific read-only registers that are available under Extended Capability ID 0x03, Multi-Bus Instance.

This capability is only required for the first Bus Controller instance (the one that starts at absolute base Address 0x0, or BAR0 for PCIe).

A Host Controller with the Multi-Bus Instance capability supports multiple I3C Buses (up to 15), with one instance of the HCI Register Set and one instance of I3C Bus Controller Logic for each I3C Bus, in a single hardware function (e.g. PCIe B/D/F). This approach makes it possible to build Host Controller hardware that supports multiple I3C Bus Controller instances, while keeping single-instance versions simple.

For a Host Controller with Multi-Bus Instance functionality, each HCI entry point (i.e., HCI Register Set) describes one I3C Bus Controller instance. All register offsets defined for a given I3C Bus Controller Instance are relative to the HCI entry point for that particular I3C Bus instance.

#### 7.7.4.1 Bus Controller Instance Count (+ 0x04)

Number of additional I3C Bus Controller instances (i.e., number of HCI Register Sets – 1).

**Table 88 Bus Controller Instance Count (BUS\_CTRL\_INSTANCE\_COUNT) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
<b>28</b> [31:4]	RESERVED	–	–	–
<b>4</b> [3:0]	INSTANCE_COUNT	R	IMPL	<b>Instance Count</b> Number of additional I3C Bus Controller instances available.

#### 7.7.4.2 Bus Controller Instance Offset (+ 0x08 + 4\*(INSTANCE\_NUMBER-1))

The Bus Controller Instance Offset is an array of registers. For each Bus Controller instance, the Offset register contains the offset from the primary/absolute HCI base (0x0) to the HCI Register Set for the implemented I3C Bus Controller instances.

##### Example:

Consider the case of a specific Host Controller implementation featuring Bus Instance Capability with:

- **Bus Controller instance 0 at offset 0x0**

Of course instance 0 must start at offset 0x0. The Multi-Bus Instance capability can skip this Offset.

- **Bus Controller instance 1 at offset 0x400**

**HCI\_VERSION** for instance 1 is thus located at offset 0x400.

- **Bus Controller instance 2 at offset 0x1200**

**HCI\_VERSION** for instance 2 is thus located at offset 0x1200.

In this example, Bus Controller instance 0 features Multi-Bus instance capability with Instance Count equal to 2, Bus Controller instance 1 offset is 0x400, and Bus Controller instance 2 offset is 0x1200. Optionally, Bus Controller instance 0 may be introduced, in which case the Instance Count equals 3 and Bus Controller instance 3 offset is 0x0.

In the title of **Table 89**, the hash characters ("##") represent the Bus Controller instance number.

**Table 89 Bus Controller Instance ## Offset (BUS\_CTRL\_INSTANCE\_##\_OFFSET) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	INSTANCE_OFFSET	R	IMPL	<b>Instance Offset</b> Offset to the HCI Register Set (register <b>HCI_VERSION</b> , or HCI entry point) for an additional Bus Controller instance. The number of Bus Controller instances shall not exceed 15.

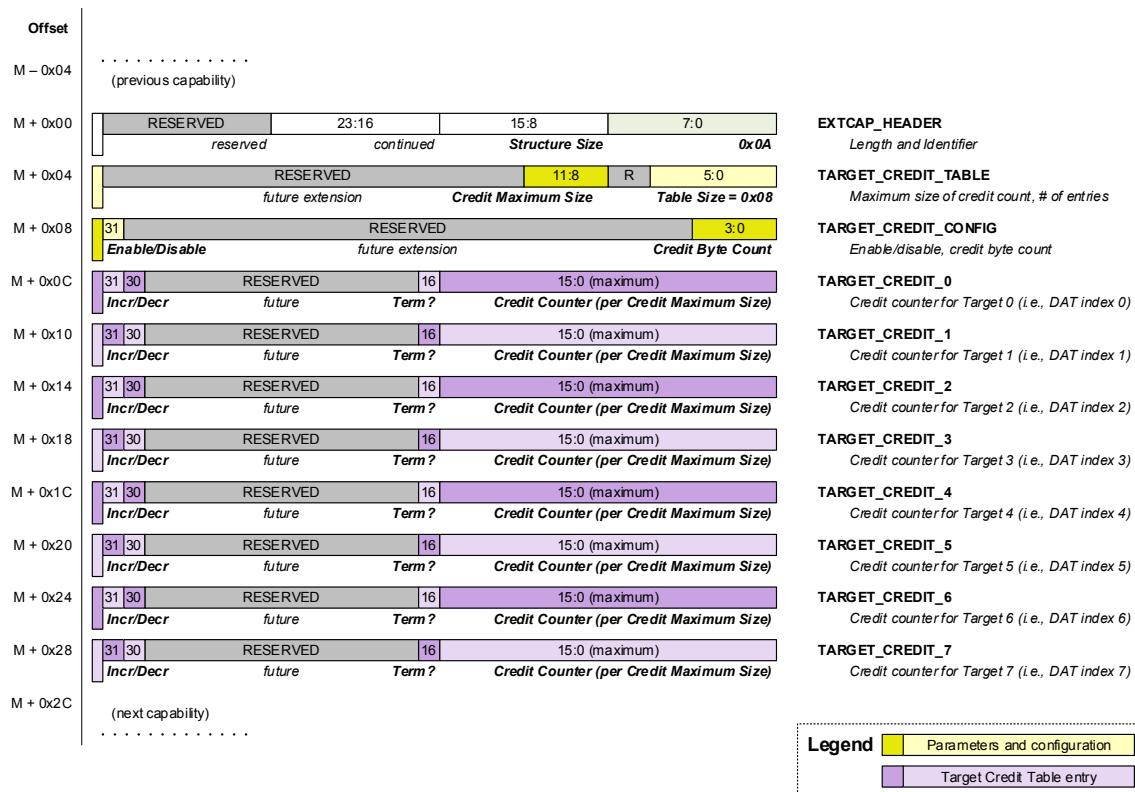
### 7.7.5 Target IBI Credit Counters (ID = 0x0A)

This Section defines the optional Capability-specific registers available under Extended Capability ID 0x0A, Target IBI Credit Counters. This capability includes a credit counting mechanism that allows individual I3C Targets to have individual credit counters for IBI payload data as well as associated Pending Read data (i.e., when used with Auto-Command, per [Section 6.11](#)). A Host Controller may support this Extended Capability; if so, then it shall implement the credit counting mechanism that manages whether individual I3C Targets are allowed to send IBI Requests on the I3C Bus, based on the credit counter for each I3C Target (see [Section 6.9.5](#)).

This Extended Capability structure includes a Target Credit Table, consisting of an array of registers with one register for each Target. The size of the Target Credit Table array shall be the size of the DAT (i.e., the same number of entries), as indicated by field **TABLE\_SIZE** in register **TARGET\_CREDIT\_TABLE** (see [Section 7.7.5.1](#)).

#### Structure Template and Examples

The following example shows how this type of Extended Capability structure could be defined for different Host Controller implementations. For these examples, the structures are shown with a starting offset **M** within the register offset map. It is assumed that this offset is within the range that register **EXT\_CAPS\_SECTION\_OFFSET** ([Section 7.7](#)) refers to, and that it can be successfully read by the Driver.



**Figure 45 Target IBI Credit Counters Structure Example**

The **Figure 45** example shows a Host Controller that supports Target IBI credit counting and has 8 entries in the Target Credit Table (i.e., because it also has 8 entries in the DAT). Each **TARGET\_CREDIT\_##** register in the Target Credit Table provides fields for updating the credit count.

### 7.7.5.1 Target Credit Table (TARGET\_CREDIT\_TABLE) (+ 0x04)

6146 This register describes the structure of the Target Credit Table, and the credit counting capabilities of the Host Controller.  
6147

6148 **Table 90 Target Credit Table (TARGET\_CREDIT\_TABLE) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
20 [31:12]	RESERVED	—	—	—
4 [11:8]	CREDIT_MAX_SIZE	R	0x0	<p><b>Target Credit Maximum Size</b>            Effective width (in bits) of field <b>CREDIT_COUNT</b> in each register of the Target Credit Table. This value determines the number of writeable bits in field <b>CREDIT_COUNT</b>, starting with bit 0.</p> <p><b>Valid Values:</b> 1–15  <b>Values:</b></p> <ul style="list-style-type: none"> <li>The value is 0-based.</li> </ul> <p><b>Examples:</b></p> <ul style="list-style-type: none"> <li>A field value of 4'd1 (the minimum) would represent 2 writeable bits (i.e., Bits[1:0], for a range of 0 to 3 credits)</li> <li>A field value of 4'd3 would represent 4 writeable bits (i.e., Bits[3:0], for a range of 0 to 15 credits)</li> <li>A field value of 4'd5 would represent 6 writeable bits (i.e., Bits[5:0], for a range of 0 to 63 credits)</li> <li>A field value of 4'd15 (the maximum) would represent 16 writeable bits (i.e., Bits[15:0], for a range of 0 to 65535 credits)</li> </ul>
2 [7:6]	RESERVED	—	—	—

Size [Bits]	Field Name	Memory Access	Reset Value	Description
6 [5:0]	TABLE_SIZE	R	IMPL	<p><b>Target Credit Table Size</b>  Number of entries in the Target Credit Table that are available for use.  This value shall match the value of field TABLE_SIZE in register DAT_SECTION_OFFSET (see <a href="#">Section 7.4.11</a>).</p> <p><b>Valid Values:</b> 1–32  <b>Values:</b></p> <ul style="list-style-type: none"> <li>• The value is 1-based.</li> </ul> <p><b>Examples:</b></p> <ul style="list-style-type: none"> <li>• A field value of 6'd1 would represent 1 entry (the minimum)</li> <li>• A field value of 6'd32 would represent 32 entries (the maximum)</li> </ul>

### 7.7.5.2 Target Credit Configuration (TARGET\_CREDIT\_CONFIG) (+ 0x08)

6149 This register is used to program the Target IBI credit mechanism, and to enable or disable the mechanism.

6150 **Table 91 Target Credit Configuration (TARGET\_CREDIT\_CONFIG) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [31]	R/S	R/W	0x0	<p><b>Target IBI Credit Enable/Disable</b>          Enables or disables the Target IBI credit mechanism. Any modification of other fields in this register shall only be performed while in <b>DISABLED</b> state.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>1'b0: DISABLED:</b> The Host Controller shall not use the Target IBI credit mechanism for incoming IBI Requests.</li> <li>• <b>1'b1: ENABLED:</b> The Host Controller shall use the Target IBI credit mechanism for incoming IBI Requests.</li> </ul>
27 [30:4]	RESERVED	—	—	—
4 [3:0]	CREDIT_BYTE_COUNT	R/W	0x1	<p><b>Target Credit Byte Count</b>          Determines the number of IBI payload data bytes per credit.          This field is read-only while the Target IBI credit mechanism is in the <b>ENABLED</b> state.</p> <p><b>Valid Values:</b> 1–9</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• The value is 0-based, and indicates the power-of-2 value in data bytes that each Target's credit represents.</li> </ul> <p><b>Examples:</b></p> <ul style="list-style-type: none"> <li>• A field value of 4'd1 (the minimum) means that the Target's credit counter decrements after every 4 (<math>2^2</math>) bytes of IBI payload data.</li> <li>• A field value of 4'd4 means that the Target's credit counter decrements after every 32 (<math>2^5</math>) bytes of IBI payload data.</li> <li>• A field value of 4'd9 (the maximum) means that the Target's credit counter decrements after every 1024 (<math>2^{10}</math>) bytes of IBI payload data.</li> </ul>

### 7.7.5.3 Target Credit Count (TARGET\_CREDIT\_##) (+ offset varies)

This register serves two purposes:

- The Driver may write to this register to update (i.e., either increment or decrement) the credit counter for a particular I3C Target; or
- The Driver may read this register to get the current credit counter for a particular I3C Target.

This register exists within an array in the Extended Capability structure, and the index corresponds with the same index in the DAT. Therefore, this array has the same number of entries as the DAT, and each Target that can raise IBI Requests will require both a DAT entry and a register in this array, with the same index. The first instance of this register is at offset 0x0C from the start of the Extended Capabilities structure.

**Note:**

*In the title of Table 92, the hash characters ("##") represent the array index.*

**Table 92 Target Credit Count ## (TARGET\_CREDIT\_##) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [31]	CREDIT_INCR	W	0x0	<p><b>Target Credit Increment</b>  A value of 1'b1 indicates that the Host Controller should increment the Target's credit counter by the amount in field <b>CREDIT_COUNT</b>.</p> <p><b>Note:</b>  <i>The Driver shall not set both this field and field CREDIT_DECR when writing to this register.</i></p>
1 [30]	CREDIT_DECR	W	0x0	<p><b>Target Credit Decrement</b>  A value of 1'b1 indicates that the Host Controller should decrement the Target's credit counter by the amount in field <b>CREDIT_COUNT</b>.</p> <p><b>Note:</b>  <i>The Driver shall not set both this field and field CREDIT_INCR when writing to this register.</i></p>
13 [29:17]	RESERVED	-	-	-
1 [16]	TERM_READ_ZERO_CREDIT	R/W	0x0	<p><b>Terminate Read on Zero Credit</b>  This field determines whether the Host Controller must terminate the data payload read if the Target's IBI credit counter goes to zero before the end of the data payload.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: NO_TERM: Do not terminate the data payload read if the credit counter goes to zero.</li> <li>• 1'b1: TERM_ZERO: Terminate the data payload read if the credit counter goes to zero.</li> </ul>

Size [Bits]	Field Name	Memory Access	Reset Value	Description
16 [15:0]	CREDIT_COUNT	R/W	0x0	<b>Target Credit Count</b> When reading this register, this field indicates the number of remaining credits for this Target. If the credit counter is 0, then the Host Controller shall NACK any IBI Requests and then send the <b>DISEC</b> Direct CCC with ' <b>DISINT</b> ' bit set to disable subsequent IBI Requests from this Target. When writing this register, this field indicates the amount to either increment the credit counter (i.e., if field <b>CREDIT_INCR</b> = 1'b1) or decrement the credit counter (i.e., if field <b>CREDIT_DECR</b> = 1'b1).

### 7.7.6 Dead Bus Recovery (ID = 0x0B)

This Section defines the optional Capability-specific registers available under Extended Capability ID 0x0B, Dead Bus Recovery. This capability includes a Dead Bus Recovery Mechanism that uses the I3C-defined Error Type DBR, which is defined as an optional recovery procedure at *Section 5.1.10.1.8* of the I3C Specification [*MIPI02*]. A Host Controller may support this Extended Capability; if so, then it shall implement support for Error Type DBR handling in its I3C Bus Controller Logic, and shall also implement sufficient other logic to test for the presence of a dead I3C Bus (i.e., when there is no Active Controller that responds to a START Request).

The Dead Bus Recovery Mechanism is intended to operate on its own, and also in cooperation with an optional Extended Capability that supports Standby Controller mode (i.e., acting as a Secondary Controller on the I3C Bus). The implementer may choose whether the Extended Capability for Standby Controller mode is implemented either (a) according to this I3C HCI Specification (i.e., per *Section 6.17* and *Section 7.7.11*), or (b) within another vendor-defined Extended Capability, as long as the Host Controller can join the I3C Bus as either an Active Controller or a Standby Controller.

The possible options for testing the I3C Bus to determine whether there is already an Active Controller (i.e., one that is responsive and active) might vary, depending on whether support for Standby Controller mode is also present.

#### 7.7.6.1 Dead Bus Recovery Engage (DBR\_ENGAGE) (+ 0x04)

The Dead Bus Recovery Engage register is a write-only register that engages the Dead Bus Recovery mechanism, and attempts to test the I3C Bus for an Active Controller (i.e., not this Host Controller) that is active and responsive. Software shall write to this register to configure the Dead Bus Recovery mechanism and arm it for engagement, i.e., when a suitable Internal Control Command is enqueued (per *Section 8.4.2.8*).

**Table 93 Dead Bus Recovery Engage (DBR\_ENGAGE) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
<b>9</b> [31:23]	RESERVED	—	—	—
<b>7</b> [22:16]	<b>DBR_IBI_ADDRESS</b>	W	0x0	<b>IBI Address</b> If field <b>DBR_MODE</b> is 3'd1, then the I3C Bus Controller Logic shall use the I3C Address in this field for an attempted In-Band Interrupt (i.e., acting as a Target) after the START Request, if the Active Controller responds to the START Request.
<b>6</b> [15:10]	RESERVED	—	—	—

Size [Bits]	Field Name	Memory Access	Reset Value	Description
2 [9:8]	DBR_ASX_TIMING	W	0x0	<p><b>Activity State Timing</b>          Specifies the timing based on the expected Activity State that the Active Controller (i.e., not this Host Controller) may have previously sent on the I3C Bus, using the ENTAS0–ENTAS3 CCCs. The Dead Bus Recovery mechanism shall use this value as the maximum time to wait for the Active Controller to respond, before claiming The Controller Role if it does not respond.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>2'd0–2'd2: Values are not legal, per I3C Specification (i.e., t<sub>CAS</sub> based on ENTAS<sub>x</sub> CCCs, where <math>x</math> is 0–2)</li> <li>2'd3: Wait for 50 ms (i.e., maximum t<sub>CAS</sub> based on ENTAS3 CCC)</li> </ul>
1 [7]	RESERVED	—	—	—
3 [6:4]	DBR_MODE	W	0x0	<p><b>Engagement Mode</b>          Specifies the action that the Host Controller shall take when engaging the Dead Bus Recovery mechanism to test the I3C Bus.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>3'd0: SDA_QUICK_RELEASE: If the Active Controller responds, then release SDA and do not drive any bits in the Arbitrable Address Header.</li> <li>3'd1: USE_IBI_ADDRESS: If the Active Controller responds, then drive the I3C Address provided in field DBR_IBI_ADDRESS.</li> <li>3'd2: USE_HJ_ADDRESS: If the Active Controller responds, then simulate a Hot-Join Request.</li> <li>3'd3: USE_CR_DEVICE_ADDR: Use valid field DYNAMIC_ADDRESS in register CONTROLLER_DEVICE_ADDR (<a href="#">Section 7.4.3</a>), if and only if that Address field is valid.</li> <li>3'd4: Use implementer-defined logic (i.e., Secondary Bus Controller Logic for Standby Controller mode) to test the I3C Bus, which automatically chooses the initial role per <a href="#">Section 6.17.1</a>.</li> <li>All other values: Reserved for future use.</li> </ul> <p>If the chosen Address wins Arbitration, then the Host Controller shall not provide a data payload for the IBI.</p>

Size [Bits]	Field Name	Memory Access	Reset Value	Description
4 [3:0]	DBR_VERIF_KEY	W	0x0	<p><b>Verification Key</b></p> <p>If software writes a value of 0xD to this field, then the Host Controller shall use the Dead Bus Recovery mechanism and shall attempt to perform the action specified in the other fields of this register.</p> <p>If software writes any other value (i.e., 0x0) to this field, then the Host Controller shall ignore values written to the other fields in this register.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>0xD: ENGAGE:</b> Verifies that other field values are valid (as intended by software)</li> <li>• All other values: Host Controller takes no action</li> </ul>

6184 **Note:**

6185     *If the Host Controller does not support Standby Controller mode, or if this mode is supported (i.e.,*  
 6186     *via vendor-defined Extended Capabilities) but not currently enabled, then field **BUS\_ENABLE** in*  
 6187     *register **HC\_CONTROL** (see **Section 7.4.2**) must be set to **DISABLED** before using the Dead Bus*  
 6188     *Recovery mechanism. This prevents software from requesting the Host Controller to test for a*  
 6189     *Dead Bus if this Host Controller is in fact the Active Controller of the Bus.*

6190     *If the test for Dead Bus Recovery does not induce a response to the START Request by the Active*  
 6191     *Controller, then the Host Controller shall automatically enable Host Controller operation in Active*  
 6192     *Controller mode (i.e., by setting field **BUS\_ENABLE** in register **HC\_CONTROL** to 1'b1) and it shall*  
 6193     *become the Active Controller, per the I3C Specification at **Section 5.1.10.1.8 [MIPI02]**.*

6194     *Refer to **Section 8.4.2.8** for the Response Descriptor that the Host Controller will generate when*  
 6195     *attempting the Dead Bus Recovery procedure using the settings in this register.*

### 7.7.7 Debug Specific (ID = 0x0C)

This Section defines the four Capability-specific read-only registers available under Extended Capability ID 0x0C, Debug Specific.

All MIPI-defined Debug specific registers shall appear in this Extended Capability.

#### 7.7.7.1 Queue Status Level (QUEUE\_STATUS\_LEVEL) (+ 0x04)

The Queue Status Level register is used to check levels of following queues:

- IBI Queue (entries, buffer count)
- Response Queue (entries)
- Command Queue (free entries)

Since queues are only used for PIO Mode, this register is only available for the PIO Mode Registers section of the Register Map. For Ring Headers, the software has access to memory pointers, and shall calculate levels on each Ring using memory pointer calculations.

**Table 94 Queue Status Level (QUEUE\_STATUS\_LEVEL) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
3 [31:29]	RESERVED	—	—	—
5 [28:24]	IBI_STATUS_CNT	R	0x0	<b>IBI Buffer Status Count</b> Number of IBI Status entries currently in the IBI Queue.
8 [23:16]	IBI_BUFFER_LVL	R	0x0	<b>IBI Buffer Level</b> Number of buffer entries currently in the IBI Queue.
8 [15:8]	RESPONSE_BUFFER_LVL	R	0x0	<b>Response Buffer Level</b> Number of buffer entries currently in the Response Queue.
8 [7:0]	CMD_QUEUE_FREE_LVL	R	IMPL	<b>Command Queue Free Buffer Level</b> Number of free buffer entries currently in the Command Queue. Reset value is the depth of the Command Queue.

### 7.7.7.2 Data Buffer Status Level (DATA\_BUFFER\_STATUS\_LEVEL) (+ 0x08)

The Data Buffer Status Level register is used to check the current levels of following queues:

- Rx Data Queue (number of entries in use)
- Tx Data Queue (number of free entries)

**Table 95 Data Buffer Status Level (DATA\_BUFFER\_STATUS\_LEVEL) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
16 [31:16]	RESERVED	—	—	—
8 [15:8]	RX_BUF_LVL	R	0x0	<b>Rx Data Buffer Status Count</b> Indicates the number of Rx Data Buffer entries in the Rx Data Queue.
8 [7:0]	TX_BUF_FREE_LVL	R	IMPL	<b>Tx Data Buffer Status Count</b> Indicates the number of free Tx Data Buffer entries in the Tx Data Queue. Reset value is the depth of the Tx Data Queue.

### 7.7.7.3 Present State Debug (PRESENT\_STATE\_DEBUG) (BASE + 0x0C)

The Present State Debug register is used to check Host Controller status. The Debug part of status register is not mandatory for operation, however it provides detailed information on internal state of the Host Controller.

**Table 96 Present State Debug (PRESENT\_STATE\_DEBUG) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
4 [31:28]	RESERVED	R	0x0	–
4 [27:24]	CMD_TID	R	0x0	<b>Command Transaction ID</b> Transaction ID of the currently executing command.
2 [23:22]	RESERVED	R	0x0	–

Size [Bits]	Field Name	Memory Access	Reset Value	Description
6 [21:16]	BCL_TFR_ST_STATUS	R	0x0	<p><b>Bus Controller Logic Transfer State Status</b></p> <p>Indicates the state of transfer that the Host Controller is currently executing.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>6'h0: IDLE:</b> Host Controller is in the Idle state, waiting either for commands from the application, or for Target-initiated In-Band Interrupts</li> <li>• <b>6'h1: START:</b> START Generation State</li> <li>• <b>6'h2: RESTART:</b> RESTART Generation State</li> <li>• <b>6'h3: STOP:</b> STOP Generation State</li> <li>• <b>6'h4: START_HOLD:</b> START Hold Generation for the Target-initiated START State</li> <li>• <b>6'h5: BCAST_WRITE:</b> Broadcast Write Address Header(7'h7E,W) Generation State</li> <li>• <b>6'h6: BCAST_READ:</b> Broadcast Read Address Header(7'h7E,R) Generation State</li> <li>• <b>6'h7: DAA:</b> Dynamic Address Assignment State</li> <li>• <b>6'h8: ADDR:</b> Target Address Generation State</li> <li>• <b>6'hB: CCC:</b> CCC Byte Generation State</li> <li>• <b>6'hC: HDR:</b> HDR Command Generation State</li> <li>• <b>6'hD: WR:</b> Write Data Transfer State</li> <li>• <b>6'hE: RD:</b> Read Data Transfer State</li> <li>• <b>6'hF: IBI_ADR_READ:</b> In-Band Interrupt Address Read Data State</li> <li>• <b>6'h10: IBI_DIS:</b> In-Band Interrupt Auto-Disable State</li> <li>• <b>6'h11: HDR_DDR_CRC:</b> HDR-DDR CRC Data Generation/Receive State</li> <li>• <b>6'h12: CLOCK_EXT:</b> Clock Extension State</li> <li>• <b>6'h13: HALT:</b> Halt State</li> <li>• <b>6'h14: IBI_READ:</b> In-Band Interrupt (IBI) Read Data State</li> </ul>
2 [15:14]	RESERVED	R	0x0	—

Size [Bits]	Field Name	Memory Access	Reset Value	Description
6 [13:8]	BCL_TFR_STATUS	R	0x0	<p><b>Bus Controller Logic Transfer Type Status</b>            Indicates the type of transfer the Host Controller is currently executing.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>6'h0: IDLE:</b> Host Controller is in the Idle state, waiting either for commands from the application, or for Target-initiated In-Band Interrupts</li> <li>• <b>6'h1: BCAST_WRITE:</b> Broadcast CCC Write Transfer</li> <li>• <b>6'h2: TARGET_WRITE:</b> Directed CCC Write Transfer</li> <li>• <b>6'h3: TARGET_READ:</b> Directed CCC Read Transfer</li> <li>• <b>6'h4: ENTDAA:</b> ENTDAA Address Assignment Transfer</li> <li>• <b>6'h5: SETDASA:</b> SETDASA Address Assignment Transfer</li> <li>• <b>6'h6: I3C_SDR_WRITE:</b> Private I3C SDR Write Transfer</li> <li>• <b>6'h7: I3C_SDR_READ:</b> Private I3C SDR Read Transfer</li> <li>• <b>6'h8: I2C_SDR_WRITE:</b> Private I<sup>2</sup>C SDR Write Transfer</li> <li>• <b>6'h9: I2C_SDR_READ:</b> Private I<sup>2</sup>C SDR Read Transfer</li> <li>• <b>6'hA: HDR_TS_WRITE:</b> Private HDR Ternary Symbol (TS) Write Transfer</li> <li>• <b>6'hB: HDR_TS_READ:</b> Private HDR Ternary Symbol (TS) Read Transfer</li> <li>• <b>6'hC: HDR_DDR_WRITE:</b> Private HDR Double-Data Rate (DDR) Write Transfer</li> <li>• <b>6'hD: HDR_DDR_READ:</b> Private HDR Double-Data Rate (DDR) Read Transfer</li> <li>• <b>6'hE: IBI:</b> Servicing In-Band Interrupt Transfer</li> <li>• <b>6'hF: HALTED:</b> Host Controller is in the Halt State, waiting for the application to resume through register <b>HC_CONTROL</b></li> </ul>
6 [7:2]	RESERVED	—	—	—
1 [1]	SDA_LINE_SIGNAL_LEVEL	R	0x1	<p><b>SDA Line Signal Level</b>            This bit is used to check the SDA Line level, in order to recover from errors and for debugging.            This bit reflects the value of synchronized <b>SDA_IN_A</b> signal.</p>

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [0]	SCL_LINE_SIGNAL_LEVEL	R	0x1	<b>SCL Line Signal Level</b> This bit is used to check the SCL Line level, in order to recover from errors and for debugging. This bit reflects the value of synchronized <b>SCL_IN_A</b> signal.

#### 7.7.7.4 Controller Error Counters (MX\_ERROR\_COUNTERS) (+ 0x10)

Controller Error Counters are maintained to allow the software to detect any anomalies on the I3C Bus.

6215 6216 **Table 97 MX Error Counters (MX\_ERROR\_COUNTERS) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
<b>24</b> [31:8]	RESERVED	—	—	—
<b>8</b> [7:0]	<b>CE2_ERROR_COUNT</b>	R	0x0	<b>CE2 Error Counter</b> Counts I3C Type CE2 errors on the I3C Bus. The Host Controller shall clear this register upon each read from the Host.

### 7.7.7.5 Scheduled Commands Debug (SCHED\_CMDS\_DEBUG) (+ 0x14)

If the Host Controller supports at least one instance of Scheduled Commands logic (per *Section 6.16*), then this register contains debug status for errors that can be generated while processing Scheduled Commands.

**Table 98 Scheduled Commands Debug (SCHED\_CMDS\_DEBUG) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
10 [31:22]	RESERVED	—	—	—
1 [21]	ERR_OCCURRED	R	0x0	<p><b>Error Occurred</b>            If this field is 1'b0, then no error is reported by this register, and all subsequent field values should be ignored.            If this field is 1'b1, then subsequent field values indicate the nature of the error.</p>
5 [20:16]	TICK_INTERVAL	R	0x0	<p><b>Tick Interval Number</b>            If an error occurred, then this field shall contain the tick interval number. The meaning depends on the value of field <b>ERR_TYPE</b>.  <b>Values for Error Type 1'b0</b>            (Transfer Command execution error):            The tick interval in which the Transfer Command caused the error.  <b>Values for Error Type 1'b1</b>            (Tick interval missed):            The tick interval that was missed.</p>
8 [15:8]	ENTITY_ID	R	0x0	<p><b>Entity ID</b>            If an error occurs, then this field shall contain the specific identifier for the scheduled entity where the error was caught, within this instance of Scheduled Commands logic.  <b>Values for Handler Type 0:</b>            Defined by the implementer  <b>Values for Handler Type 1:</b>            0x0 through 0xFF: Entry identifier in the array of Schedule Table Entries (i.e., from 0 to <b>MAX_COUNT</b>)  <b>Values for Handler Type 2:</b>            0x0 through 0xFF: Target identifier (i.e., within register array <b>SCHEDULE_SEQ_TARGETS_##</b>)  <b>Values for Handler Type 3:</b>            0x0 through 0xFF: Index of the Transfer Descriptor in the Schedule Buffer  <b>Values for other Handler Types:</b>            Reserved for future definition</p>

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [7]	ERR_TYPE	R	0x0	<p><b>Error Type</b>            If an error occurs, then this field shall contain the reason that the error was generated.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>1'b0: Transfer Command execution error</li> <li>1'b1: Tick interval missed</li> </ul>
3 [6:4]	INST_ID	R	0x0	<p><b>Instance ID</b>            If an error occurs, then this field shall contain the instance ID for the instance of Scheduled Commands logic that generated the error.</p> <p><b>Values for Handler Type 0:</b>            Defined by the implementer</p> <p><b>Values for Handler Type 1:</b>            3'd0 always</p> <p><b>Values for Handler Type 2:</b>            Field HT2_INST_ID in register <b>SCHEDULE_LAYOUT</b> (see Section 7.7.9.2).</p> <p><b>Values for Handler Type 3:</b>            Field HT3_INST_ID in register <b>SCHEDULE_LAYOUT</b> (see Section 7.7.10.2).</p> <p><b>Values for other Handler Types:</b>            Reserved for future definition</p> <p><b>Note:</b>  <i>Handler Type 1 does not have an instance ID.</i></p>
4 [3:0]	SCHED_HANDLER	R	0x0	<p><b>Scheduled Command Handler Type</b>            If an error occurs, then this field shall contain the value in field <b>SCHED_HANDLER</b> in register <b>SCHEDULE_CAPABILITIES</b> for the instance of Scheduled Commands logic that generated the error. See <b>Table 8</b>.</p>

### 7.7.8 Scheduled Commands for Handler Type 1 (ID = 0x0D)

This Section defines the Capability-specific registers available under Extended Capability ID 0x0D, Scheduled Commands for Handler Type 1 (Simple Table). The Scheduled Commands capability is an optional capability that allows the Host Controller to issue certain Transfer Commands on a programmed schedule (see [Section 6.16](#)) while operating in Active Controller mode. The Host Controller uses registers to define the Scheduled Commands data structure that contains Command Descriptors and other configuration parameters relating to this instance of Scheduled Commands capability and its logic within the Host Controller.

**Note:**

*This Extended Capability structure only applies to Handler Type 1 (Simple Table), as defined in [Section 6.16.2](#).*

The data structure shall contain, in the following order:

- A **SCHEDULE\_CAPABILITIES** register ([Section 7.7.8.1](#)) that indicates the basic timing parameters and clock resolution as well as the size of the table of unique entries that might be configured by software for unique Transfer Commands.
- A **SCHEDULE\_TABLE\_LAYOUT** register ([Section 7.7.8.2](#)) that defines this structure for Handler Type 1, and describes the sizes and layout of the Schedule Table Entries (i.e., data structures) available to software.
- A **SCHEDULE\_CONFIG** register ([Section 7.7.8.3](#)) that contains run-time control over the operation of this instance of Scheduled Commands logic and the number of entries that are enabled for programmed execution.

These first three registers are required, and shall serve as a descriptor for the Extended Capability structure residing at a fixed offset relative to the starting offset for the structure (i.e., the instance of **EXTCAP\_HEADER**).

After these entries, the data structure shall contain a contiguous set of Schedule Table Entries arranged in an array. Each such array entry shall consist of a set of the following registers, identified by index “##”:

- Registers **SCHEDULE\_TABLE\_##\_MASK** ([Section 7.7.8.4](#)) and **SCHEDULE\_TABLE\_##\_CONFIG** ([Section 7.7.8.5](#)). These registers shall be used to indicate the specific clock ticks at which the specific Transfer Command shall be executed, as well as how often the Transfer Command shall be executed (i.e., only execute on every **N**-th cycle of ticks, if supported).
- Registers to contain one Command Descriptor (i.e., **SCHEDULE\_TABLE\_##\_COMMAND**, [Section 7.7.8.6](#)) to be executed according to the criteria above. The Command Descriptor shall be stored in an appropriate number of registers for the Host Controller’s supported Command Descriptor format (see [Section 6.7](#) and [Section 8.4](#)).

In this section, the index for such Schedule Table Entries may be in the range between 0 and the value of field **MAX\_COUNT** in register **SCHEDULE\_CAPABILITIES**. For simple configurations, this Extended Capability structure may contain a fixed number (i.e., up to 16) of Schedule Table Entries. For more advanced configurations, this Extended Capability structure might be defined with other variants based on alternative use models. Field **SCHED\_HANDLER** in register **SCHEDULE\_TABLE\_LAYOUT** defines the overall format of the table and its registers.

**Note:**

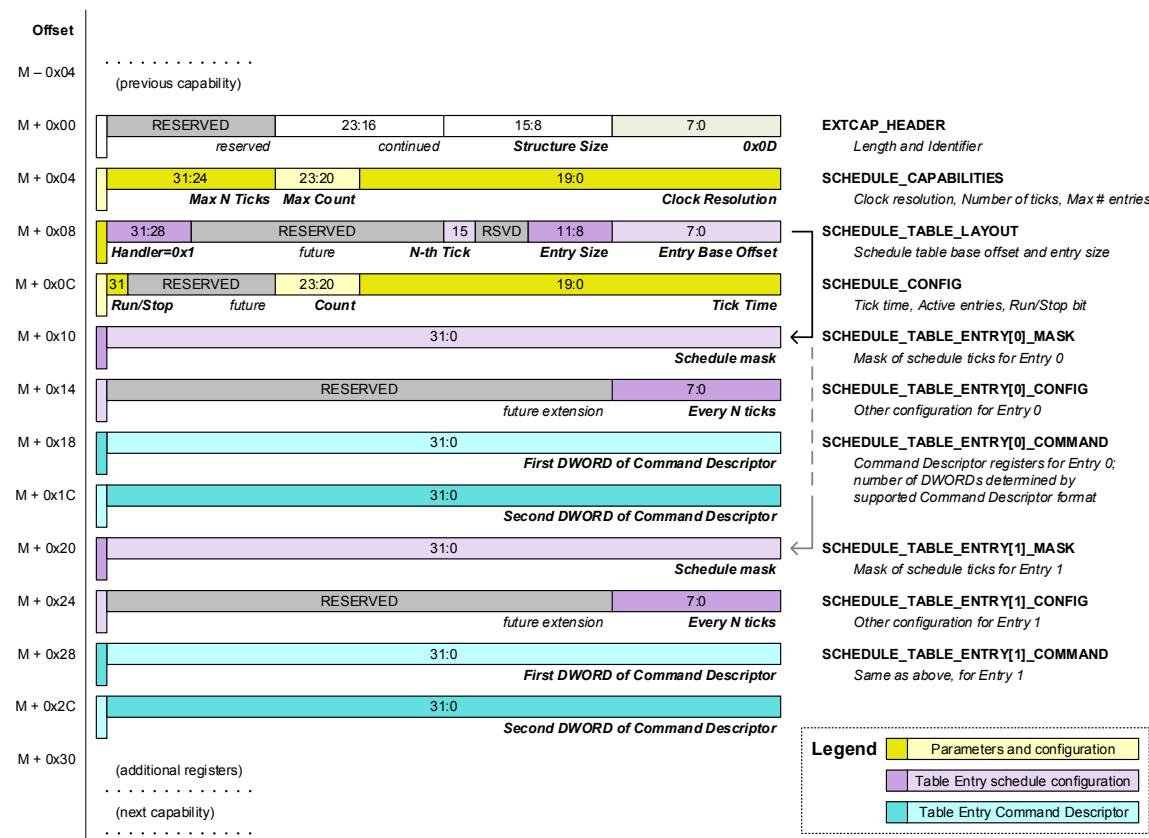
*Subsequent text and register field tables assume a handler method value of 0x1, where each Schedule Table Entry contains one Command Descriptor.*

Since the size of the Command Descriptor depends on the supported Command Descriptor format, both software and the Host Controller’s Scheduled Commands logic must understand the correct relationship, based on the entry sizes indicated in register **SCHEDULE\_TABLE\_LAYOUT**, in order to determine the location and layout of the table. Note that the entry size for the Schedule Table Entries might be larger than the required size of the sum of registers for schedule configuration and for the Command Descriptor, in order

to allow room for additional implementer-specific configuration or future extension. The size of the Schedule Table Entry is indicated by field **ENTRY\_SIZE** in register **SCHEDULE\_TABLE\_LAYOUT**.

### Structure Template and Examples

The following examples show how this type of Extended Capability structure could be defined for several different Host Controller implementations. For these examples, the structures are shown with a starting offset **M** within the register offset map. It is assumed that this offset is within the range that register **EXT\_CAPS\_SECTION\_OFFSET** (*Section 7.7*) refers to, and that it can be successfully read by the Driver.



**Figure 46 Scheduled Commands Structure Example (Handler Type 1)**

The **Figure 46** example shows a Host Controller that supports Scheduled Commands with a single Command Descriptor in each entry. Each Schedule Table Entry is four DWORDs in size, and holds two DWORD registers for the single Command Descriptor, since the Command Descriptor size is two DWORDs and is stored directly in the entry.

Additional implementation choices are possible, including the following configuration options:

- The size of the array of Scheduled Table Entries might be increased, up to the maximum size of 16 entries.
- The Schedule Table Entries might start at a higher byte offset than 0x10, indicated by different values of field **ENTRY\_BASE\_OFFSET** (i.e., values greater than 0x2), thereby leaving space for additional configuration registers that might be present after register **SCHEDULE\_CONFIG**, and before the first instance of register **SCHEDULE\_TABLE\_ENTRY\_##\_MASK**, to allow for future extension.
- The Schedule Table Entry size might be larger than four DWORDs, indicated by different values of field **ENTRY\_SIZE**, to allow for additional per-entry registers that could store additional implementer-specific configuration or be used for future extension with handler method 0x0 or some other handler method (i.e., not yet defined).

- 6290
- 6291
- 6292
- For this option, note that the relative starting offsets of the required registers within each table entry would remain unchanged, and any additional per-entry registers would follow the registers that stored the Command Descriptor.

### 7.7.8.1 Schedule Capabilities (SCHEDULE\_CAPABILITIES) (+ 0x04)

This register is used to provide information on the Host Controller's capabilities relating to the Scheduled Commands logic, including clock resolution and the maximum number of Commands that can be scheduled.

**Table 99 Schedule Capabilities (SCHEDULE\_CAPABILITIES) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
8 [31:24]	<b>MAX_N_TICKS</b>	R	IMPL	<p><b>Max N Ticks</b>            Hardware may implement a mechanism to support execution on every N-th opportunity. This value defines the maximum supported value for N.</p> <p><b>Valid Values:</b> 0–255  <b>Values:</b></p> <ul style="list-style-type: none"> <li>The value is 0-based.            Examples:               <ul style="list-style-type: none"> <li>A field value of 8'd1 would represent 2 ticks (i.e., 1 ≤ N ≤ 2)</li> <li>A field value of 8'd4 would represent 5 ticks (i.e., 1 ≤ N ≤ 5)</li> </ul> </li> <li><b>0:</b> Execution every N-th opportunity is not supported.</li> </ul>
4 [23:20]	<b>MAX_COUNT</b>	R	IMPL	<p><b>Scheduled Command Table Max Count</b>            Number of entries in the Scheduled Command Table that are available for use.</p> <p><b>Valid Values:</b> 0–15  <b>Values:</b></p> <ul style="list-style-type: none"> <li>The value is 0-based.            Examples:               <ul style="list-style-type: none"> <li>A field value of 4'd0 would represent 1 entry (the minimum)</li> <li>A field value of 4'd3 would represent 4 entries</li> <li>A field value of 4'd15 would represent 16 entries (the maximum)</li> </ul> </li> </ul>
20 [19:0]	<b>CLOCK_RESOLUTION</b>	R	IMPL	<p><b>Schedule Clock Resolution</b>            Number of clock ticks per 1 ms.</p> <p><b>Valid Values:</b> 0–1,048,575</p>

### 7.7.8.2 Schedule Table Layout (SCHEDULE\_TABLE\_LAYOUT) (+ 0x08)

This register defines the size of each Schedule Table Entry in this Schedule Table, as well as the starting offset of the array of Schedule Table Entries within the Extended Capability structure. This register also provides a field for the method of handling and storing the Schedule Table Entries, which allows for this structure to be extensible for future variants (i.e., for other Handler Types or for other implementations of Scheduled Commands logic).

**Table 100 Schedule Table Layout (SCHEDULE\_TABLE\_LAYOUT) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
4 [31:28]	<b>SCHED_HANDLER</b> <i>Applies to all Entries in this Schedule Table</i>	R	IMPL	<p><b>Schedule Handler</b> Defines the method of handling Schedule Table Entries in this Schedule Table, as well as the format and layout of subsequent registers (i.e., relative offset 0x10 and above).</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>0x1:</b> Each Schedule Table Entry contains a single 2-DWORD Command Descriptor</li> <li>• <b>All Other Values:</b> Either defined for other Handler Types, or reserved for future use</li> </ul>
12 [27:16]	RESERVED	-	-	-
1 [15]	<b>NTH_TICK_SUPPORTED</b>	R	IMPL	<p><b>Every N-th Tick Supported</b> Defines whether this Scheduled Commands handler supports command execution on every <b>N</b>-th opportunity.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>1'b1:</b> Supported; field <b>N_TICKS</b> in register <b>SCHEDULE_TABLE_##_CONFIG</b> is writeable</li> <li>• <b>1'b0:</b> Not supported; field <b>N_TICKS</b> in register <b>SCHEDULE_TABLE_##_CONFIG</b> is reserved</li> </ul>
3 [14:12]	RESERVED	-	-	-

Size [Bits]	Field Name	Memory Access	Reset Value	Description
4 [11:8]	<b>ENTRY_SIZE</b> <i>Applies to all Entries in this Schedule Table</i>	R	IMPL	<p><b>Schedule Table Entry Size</b>  Size in DWORDs of each Schedule Table Entry in this Schedule Table. The minimum size of the Schedule Table Entry shall accommodate one Command Descriptor plus the two per-entry registers (see <a href="#">Section 7.7.8.4</a> and <a href="#">Section 7.7.8.5</a>).</p> <p><b>Note:</b> The field is zero-based, and the Entry size in DWORDS is <math>2 \times (\text{the field value} + 1)</math>.</p> <p><b>Valid Values:</b> 4'd0–4'd5 (2–12 DWORDs), depending on the implementer-defined number of other registers per Schedule Table Entry</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 4'd0: Reserved for future use: (2 DWORDs per Entry is the minimum size for an Entry with no Command Descriptor)</li> <li>• 4'd1: 4 DWORDs per Entry (this is the minimum size for an Entry with a single 2-DWORD Command Descriptor)</li> <li>• 4'd2: 6 DWORDs per Entry</li> <li>• ...</li> <li>• 4'd5: 12 DWORDs per Entry</li> </ul>
8 [7:0]	<b>ENTRY_BASE_OFFSET</b>	R	IMPL	<p><b>Schedule Table Base Offset</b>  Base Offset of the array of Schedule Table Entries, in DWORDs, relative to the offset of this register.</p> <p>A value of 0x2 indicates that the array starts at relative offset 0x10 within the Extended Capability structure, and immediately follows register <b>SCHEDULE_CONFIG</b>.</p> <p><b>Valid Values:</b> 2–255</p>

### 7.7.8.3 Schedule Configuration (SCHEDULE\_CONFIG) (+ 0x0C)

6303 This register is used to program the Schedule timing in terms of ticks (at the Schedule Clock resolution),  
 6304 and to run or stop the execution of Scheduled Commands.

6305 **Table 101 Schedule Configuration (SCHEDULE\_CONFIG) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [31]	R/S	R/W	0x0	<p><b>Scheduled Command Run/Stop</b>            Enables or disables processing for this instance of Scheduled Commands logic.            Any modification to the Schedule Table registers shall only be performed while in <b>STOPPED</b> state.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: <b>STOPPED</b>: The Host Controller shall not run the Scheduled Commands schedule</li> <li>• 1'b1: <b>ENABLED</b>: The Host Controller shall run the Scheduled Commands schedule</li> </ul>
7 [30:24]	RESERVED	-	-	-
4 [23:20]	COUNT	R/W	-	<p><b>Scheduled Command Table Count</b>            Number of active entries in the Scheduled Command Table that are enabled for execution.            Software may write to this field to limit the number of enabled entries.            Software may not write a value larger than field <b>MAX_COUNT</b> in register <b>SCHEDULE_CAPABILITIES</b> (<a href="#">Section 7.7.8.1</a>).  <b>Valid Values:</b> 0–MAX_COUNT  <b>Values:</b></p> <ul style="list-style-type: none"> <li>• The value is 0-based.            Examples:               <ul style="list-style-type: none"> <li>• A field value of 4'd0 would mean 1 active entry in the Scheduled Command Table</li> <li>• A field value of 4'd5 would mean 6 active entry in the Scheduled Command Table</li> </ul> </li> </ul>

Size [Bits]	Field Name	Memory Access	Reset Value	Description
20 [19:0]	TICK_TIME	R/W	–	<b>Tick Time</b> Indicates the number of clock cycles that are used to define the tick interval (per <b>Section 6.16.1</b> ). If the value of this field is equal to the value of field <b>CLOCK_RESOLUTION</b> (in register <b>SCHEDULE_CAPABILITIES</b> , <b>Section 7.7.8.1</b> ), then the tick interval shall be 1 ms. <b>Valid Values:</b> 0–1,048,575

#### 7.7.8.4 Schedule Table Entry Mask (SCHEDULE\_TABLE\_##\_MASK) (+ offset P varies)

This register defines the bit mask to be applied (using bitwise logical AND) to the schedule tick interval number when determining when to execute the Command Descriptor in the corresponding Command Descriptor entry.

Since the location and size of the Schedule Table Entries within the Extended Capability structure can vary, the offset of each instance of this register might also vary. The following formula gives the offset in DWORDs within the Extended Capability structure relative to the array index for the entry, where the index can range from 0 to **MAX\_COUNT**:

$$\text{Offset} = (2 + \text{ENTRY\_BASE\_OFFSET} + (\text{sizeof(ENTRY\_SIZE}) \times \text{INDEX}))$$

**Note:**

Field **MAX\_COUNT** is found in register **SCHEDULE\_CAPABILITIES** for this Schedule Table ([Section 7.7.8.1](#)). Fields **ENTRY\_BASE\_OFFSET** and **ENTRY\_SIZE** are found in register **SCHEDULE\_TABLE\_LAYOUT** for this Schedule Table ([Section 7.7.8.2](#)).

For the lowest possible value of field **ENTRY\_BASE\_OFFSET** (i.e., 0x2), the first instance of this register (i.e., index 0 in the Schedule Table Entries array) will reside at relative offset 0x10 bytes from the start of the Extended Capability structure. Each subsequent instance of this register would start at a further offset increment based on the Schedule Table Entry size, as indicated by field **ENTRY\_SIZE** (in register **SCHEDULE\_TABLE\_LAYOUT**).

**Note:**

In the title of [Table 102](#), the hash characters ("##") represent the Schedule Table entry index.

**Table 102 Schedule Table Entry ## Mask (SCHEDULE\_TABLE\_##\_MASK) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	<b>SCHEDULE_MASK</b>	R/W	–	<p><b>Schedule Information</b>            Bit mask to apply to the schedule tick interval number when determining when the associated Command is to be executed.            The Host Controller shall process this register one bit at a time, starting from Bit[0].</p> <p><b>Valid Values:</b> 0x0–0xFFFFFFFF, limited by the number of ticks supported by the Scheduled Commands logic.</p>

### 7.7.8.5 Schedule Table Entry Config (SCHEDULE\_TABLE\_##\_CONFIG) (P + 0x4)

If the Host Controller supports Scheduled Commands on every  $N$ -th opportunity, then this register's field **N\_TICKS** defines the value of  $N$ . If  $N$  is greater than 1, then the Host Controller shall only process the schedule on the  $N$ -th iteration (i.e., shall treat the schedule as empty for the first  $(N - 1)$  tick horizons).

Since the location and size of the Schedule Table Entries within the Extended Capability structure can vary, the offset of each instance of this register might also vary. The following formula gives the offset in DWORDs within the Extended Capability structure relative to the array index for the entry, where the index can vary from 0 to **MAX\_COUNT**:

$$\text{Offset} = (3 + \text{ENTRY\_BASE\_OFFSET} + (\text{sizeof(ENTRY\_SIZE}) \times \text{INDEX}))$$

**Note:**

Field **MAX\_COUNT** is found in register **SCHEDULE\_CAPABILITIES** for this Schedule Table (Section 7.7.8.1). Fields **ENTRY\_BASE\_OFFSET** and **ENTRY\_SIZE** are found in register **SCHEDULE\_TABLE\_LAYOUT** for this Schedule Table (Section 7.7.8.2).

For the lowest possible value of field **ENTRY\_BASE\_OFFSET** (i.e., 0x2), the first instance of this register (i.e., index 0 in the Schedule Table Entries array) will reside at relative offset 0x14 bytes from the start of the Extended Capability structure. Each subsequent instance of this register would start at a further offset increment based on the Schedule Table Entry size, as indicated by field **ENTRY\_SIZE** (in register **SCHEDULE\_TABLE\_LAYOUT**).

**Note:**

In the title of Table 103, the hash characters ("##") represent the Schedule Table entry index.

**Table 103 Schedule Table Entry ## Config (SCHEDULE\_TABLE\_##\_CONFIG) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
24 [31:8]	RESERVED	-	-	-

Size [Bits]	Field Name	Memory Access	Reset Value	Description
8 [7:0]	N_TICKS	R/W	–	<p><b>Schedule Information</b>  The value programmed in this field defines <math>N</math>.  This field is only valid if the HW supports scheduling the command on every <math>N</math>-th opportunity, and field <b>NTH_TICK_SUPPORTED</b> in register <b>SCHEDULE_TABLE_LAYOUT</b> indicates that this register is implemented.</p> <p><b>Valid Values:</b> 0–255  <b>Values:</b></p> <ul style="list-style-type: none"> <li>• The value is 0-based.  Examples: <ul style="list-style-type: none"> <li>• A field value of 8'd0 (<math>N=1</math>) would mean the associated Command would execute at every opportunity</li> <li>• A field value of 8'd4 (<math>N=5</math>) would mean the associated Command would execute at every 5<sup>th</sup> opportunity</li> </ul> </li> </ul>

### 7.7.8.6 Schedule Table Entry Command Descriptor (**SCHEDULE\_TABLE\_##\_COMMAND**) (P + 0x8, etc.)

Each Schedule Table Entry shall include registers containing the command(s) to be executed at the scheduled time(s). The number of registers shall be appropriate to store one complete Command Descriptor in the format supported by the Host Controller, as indicated by field **CMD\_SIZE** in register **HC\_CAPABILITIES** ([Section 7.4.4](#)).

**Note:**

*The Schedule Table Entry Command Descriptor registers shall apply to handler method 0x0, which includes a single Command Descriptor with the Schedule Table Entry. In order to support this handler method, the Host Controller's Schedule Table Entries must have a structure size with a minimum size of 4 DWORDs.*

Since the location and size of the Schedule Table Entries within the Extended Capability structure can vary, the offset of each instance of this register might also vary. The following formulas give the offset in DWORDs within the Extended Capability structure relative to the array index for the entry, where the index can vary from 0 to **MAX\_COUNT**:

Offset of first DWORD = ( 4 + **ENTRY\_BASE\_OFFSET** + ( sizeof(**ENTRY\_SIZE**) × INDEX ) )  
Offset of second DWORD = ( 5 + **ENTRY\_BASE\_OFFSET** + ( sizeof(**ENTRY\_SIZE**) × INDEX ) )  
Etc.

**Note:**

*Field **MAX\_COUNT** is found in register **SCHEDULE\_CAPABILITIES** for this Schedule Table ([Section 7.7.8.1](#)). Fields **ENTRY\_BASE\_OFFSET** and **ENTRY\_SIZE** are found in register **SCHEDULE\_TABLE\_LAYOUT** for this Schedule Table ([Section 7.7.8.2](#)).*

Within each Schedule Table Entry starting at DWORD offset *E*, the Command Descriptor registers will start at DWORD *E*+2 (i.e., byte offset 0x8) and continue in order, to contain all DWORDs for the single Command Descriptor. Note that DWORDs *E*+0 and *E*+1 hold the other per-Entry registers (see [Section 7.7.8.4](#) and [Section 7.7.8.5](#)).

Each of these two contiguous registers shall hold one DWORD of the Command Descriptor:

- DWORD *E*+2 shall hold the first quarter (i.e. the first DWORD, Bits[31:0]); and
- DWORD *E*+3 shall hold the second quarter (i.e. the second DWORD, Bits[63:32]).

For the lowest possible value of field **ENTRY\_BASE\_OFFSET** (i.e., 0x2), the first register for the first instance of a Command Descriptor (i.e., for array index 0 in the array of Schedule Table Entries) will reside at relative offset 0x18 bytes from the start of the Extended Capability structure. Each subsequent instance of this first register for a Command Descriptor would have an offset increment based on the size of the Schedule Table Entry, as indicated by field **ENTRY\_SIZE** (in register **SCHEDULE\_TABLE\_LAYOUT**).

The Command Descriptor is defined in [Section 8.4](#).

**Note:**

*The hash characters ("##") represent the Schedule Table Entry index.*

After executing a Transfer Command from a Schedule Table Entry, the Host Controller shall generate an IBI Status Descriptor (see [Section 8.6.5](#)) followed by IBI Data Chunks (as appropriate for the Transfer Command) to indicate the status of executing the Transfer Command.

In this IBI Status Descriptor:

- Field **STATUS\_TYPE** shall be set to 3'b010.
- Field **ERROR** shall be set to 1'b0 if the Transfer Command completed successfully, or 1'b1 if there was an error. For an error, the Host Controller may continue with the next Schedule Table Entry at the appropriate time, or it may stop execution entirely and report an interrupt to the Host (via field **HC\_INTERNAL\_ERR\_STAT** in register **INTR\_STATUS**, [Section 7.4.7](#)).

- 6390     • Field **IBI\_ID** shall be set to a value that indicates the Transfer Command was executed by Scheduled  
6391       Commands logic with Handler Type 1. Bits[11:8] shall indicate the Schedule Table Entry index, and  
6392       Bits[15:12] shall be set to 1'b0.

### 7.7.9 Scheduled Commands for Handler Type 2 (ID = 0x0E)

This Section defines the Capability-specific registers available under Extended Capability ID 0x0E, Scheduled Commands for Handler Type 2 (Command Sequence). The Scheduled Commands capability is an optional capability that allows the Host Controller to issue certain Transfer Commands on a programmed schedule (see [Section 6.16](#)) while operating in Active Controller mode. The Host Controller uses registers to define the Scheduled Commands data structure that contains Command Descriptors and other configuration parameters relating to this instance of Scheduled Commands capability and its logic within the Host Controller.

**Note:**

*This Extended Capability structure only applies to Handler Type 2 (Command Sequence), as defined in [Section 6.16.3](#).*

The data structure shall contain, in the following order:

- A **SCHEDULE\_CAPABILITIES** register ([Section 7.7.9.1](#)) that indicates the basic timing parameters and clock resolution that software can configure for unique Transfer Commands.
- A **SCHEDULE\_LAYOUT** register ([Section 7.7.9.2](#)) that defines this structure for Handler Type 2, and describes the layout of the Schedule Sequence (i.e., data structure) available to software.
- A **SCHEDULE\_CONFIG** register ([Section 7.7.9.3](#)) that contains run-time control over the operation of this instance of Scheduled Commands logic.

These first three registers are required, and shall serve as a descriptor for the Extended Capability structure residing at a fixed offset relative to the starting offset for the structure (i.e., the instance of **EXTCAP\_HEADER**).

After these, the data structure shall contain arrays of registers that define the indicated Target Device(s) that will be used to execute the Command Descriptors, and the Command Descriptors to be executed. Registers that are defined in arrays are identified by index “##”:

- Register **SCHEDULE\_SEQ\_MASK** ([Section 7.7.9.4](#)) indicates the specific clock ticks that will activate the execution of the command sequence for all configured Targets.
- Register **SCHEDULE\_SEQ\_CONFIG** ([Section 7.7.9.5](#)) indicates how often the sequence of Transfer Commands shall be executed (i.e., only execute on every **N**-th cycle of ticks, if supported), and also the mask of which specific fields in subsequent registers contain valid Targets that will be used for executing the command sequence.
  - If the Host Controller is in DMA Mode, then this register also holds the Ring Bundle ID for the IBI Ring Pair to be used for any IBI Status Descriptors and optional IBI Data Chunks that will be generated as a result of processing Transfer Commands.
- Registers **SCHEDULE\_SEQ\_TARGETS\_##** ([Section 7.7.9.6](#)), with index from 0–3, allow the Driver to configure the command sequence to execute on specific Targets. This array has a minimum size of 1 register (i.e., up to 4 Targets) and a maximum size of 4 registers (i.e., up to 16 Targets). Note that the previous register **SCHEDULE\_SEQ\_CONFIG** will determine which fields in these registers are valid (i.e., will be used during execution iterations).
- An additional array of registers contains one or more **SCHEDULE\_SEQ\_COMMAND\_##** Command Descriptors ([Section 7.7.9.7](#)), to be executed according to the criteria above. Each Command Descriptor shall be stored in an appropriate number of registers for the Host Controller’s supported Command Descriptor format (see [Section 6.7](#) and [Section 8.4](#)). The minimum size of this array is 2 Command Descriptors, and the maximum size of this array is 8 Command Descriptors, as indicated by field **CMD\_SEQ\_LENGTH** in register **SCHEDULE\_LAYOUT**.

The implementer may determine the sizes of the arrays, based on the use case. Field **SCHED\_HANDLER** in register **SCHEDULE\_LAYOUT** defines the overall format of the table and its registers.

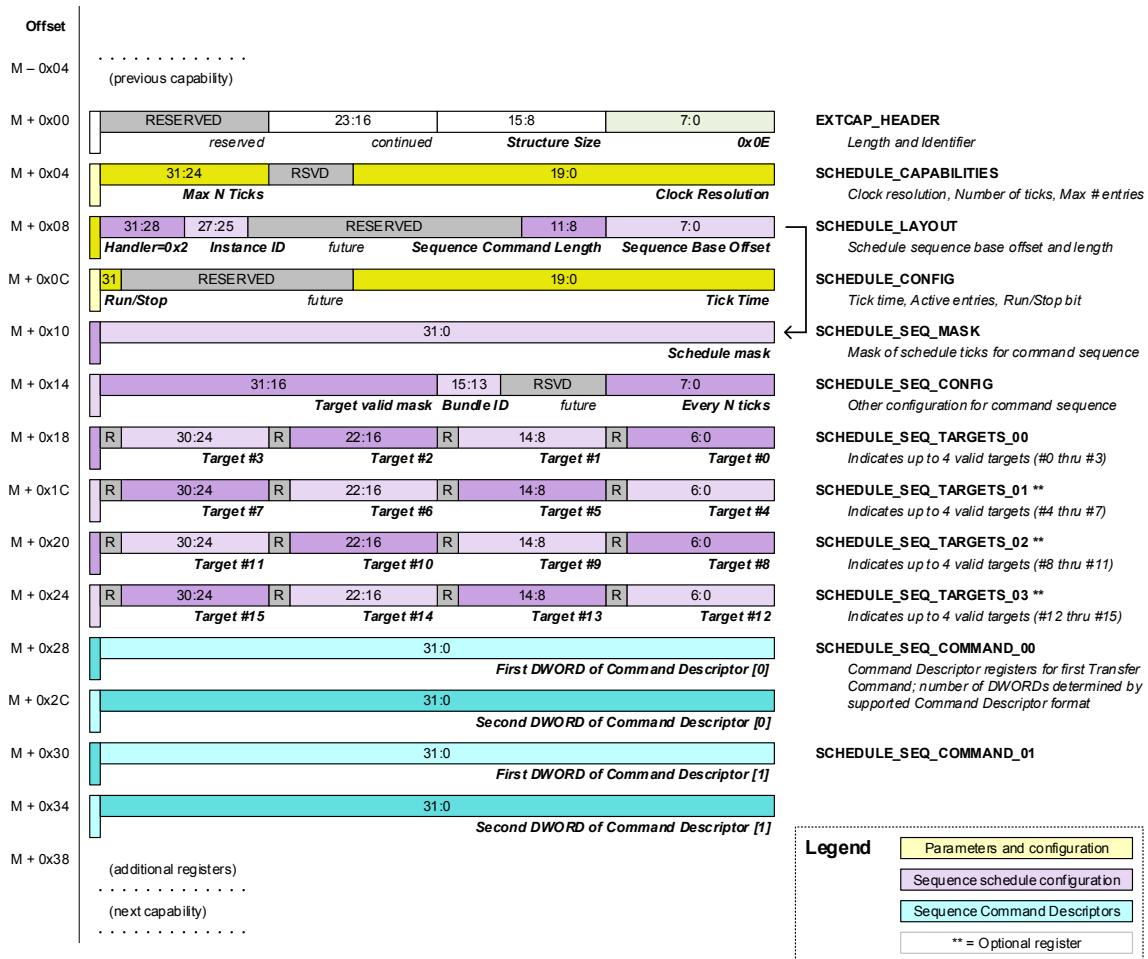
**Note:**

*Subsequent text and register field tables assume a handler method value of 0x2, which defines this structure for a command sequence and allows multiple Targets to be configured for execution.*

Since the size of the Command Descriptor depends on the supported Command Descriptor format, both software and the Host Controller's Scheduled Commands logic must understand the correct relationship, based on the layout, as indicated in register **SCHEDULE\_LAYOUT**, in order to determine the location and layout of the registers.

#### Structure Template and Examples

The following examples show how this type of Extended Capability structure could be defined, for several different Host Controller implementations. For these examples, the structures are shown with a starting offset **M** within the register offset map. It is assumed that this offset is within the range that register **EXT\_CAPS\_SECTION\_OFFSET** (*Section 7.7*) refers to, and that it can be successfully read by the Driver.



**Figure 47 Scheduled Commands Structure Example (Handler Type 2)**

The **Figure 47** example shows a Host Controller that supports Scheduled Commands with up to 16 possible Targets, and with registers that can hold a sequence of at least 2 Command Descriptors to execute for each valid Target.

6454 Additional implementation choices are possible, including the following configuration options:  
6455 • The size of the array of Command Descriptors might be increased, up to the maximum size of 8.  
6456 • The size of the array of indicated Targets might be decreased, in multiples of 4 Targets per register.  
6457 However, in this case the unused registers (i.e., **SEQUENCE\_SEQ\_TARGETS\_03** and so on) would then be  
6458 read-only registers with zero values, and the structure's layout would not otherwise change (i.e.,  
6459 **SCHEDULE\_SEQ\_COMMAND\_00** would still reside at offset 0x28, per the example above). The writeable  
6460 bits in field **TARGET\_MASK** would also be decreased to match the actual number of writeable Target  
6461 fields.  
6462 • The Schedule Sequence registers might start at a higher byte offset than 0x10, indicated by different  
6463 values of field **SEQ\_BASE\_OFFSET** (i.e., values greater than 0x2), thereby leaving space for additional  
6464 configuration registers that might be present after register **SCHEDULE\_CONFIG**, and before register  
6465 **SCHEDULE\_SEQ\_MASK**.

### 7.7.9.1 Schedule Capabilities (SCHEDULE\_CAPABILITIES) (+ 0x04)

This register is used to provide information on the Host Controller's capabilities relating to the Scheduled Commands logic, including clock resolution and the maximum number of Commands that can be scheduled.

**Table 104 Schedule Capabilities (SCHEDULE\_CAPABILITIES) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
8 [31:24]	<b>MAX_N_TICKS</b>	R	IMPL	<p><b>Max N Ticks</b>            Hardware may implement a mechanism to support execution on every <math>N</math>-th opportunity. This value defines the maximum supported value for <math>N</math>.</p> <p><b>Valid Values:</b> 0–255  <b>Values:</b></p> <ul style="list-style-type: none"> <li>The value is 0-based.            Examples:               <ul style="list-style-type: none"> <li>A field value of 8'd1 would represent 2 ticks (i.e., <math>1 \leq N \leq 2</math>)</li> <li>A field value of 8'd4 would represent 5 Ticks (i.e., <math>1 \leq N \leq 5</math>)</li> </ul> </li> <li><b>0:</b> Execution every <math>N</math>-th opportunity is not supported.</li> </ul>
4 [23:20]	RESERVED	—	—	—
20 [19:0]	<b>CLOCK_RESOLUTION</b>	R	IMPL	<p><b>Schedule Clock Resolution</b>            Number of clock ticks per 1 ms.</p> <p><b>Valid Values:</b> 0–1,048,575</p>

### 7.7.9.2 Schedule Layout (SCHEDULE\_LAYOUT) (+ 0x08)

6470 This register defines the starting offset of the Scheduled Sequence registers within the Extended Capability  
 6471 structure, as well as the number of Command Descriptors that can be stored in its registers. This register  
 6472 also allows for this structure to be extensible for future variants (i.e., for other Handler Types or for other  
 6473 implementations of Scheduled Commands logic).

6474 **Table 105 Schedule Layout (SCHEDULE\_LAYOUT) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
4 [31:28]	SCHED_HANDLER	R	IMPL	<p><b>Schedule Handler</b>            Defines the layout of subsequent registers (i.e., relative offset 0x10 and above).  <b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>0x2:</b> Single sequence of Command Descriptors</li> <li>• All other values are either defined for other Handler Types, or reserved for future use</li> </ul>
3 [27:25]	HT2_INST_ID	R	IMPL	<p><b>Instance ID for Handler Type 2</b>            This shall be a unique value between 3'd0 and 3'd7. If the Host Controller contains multiple Extended Capability structures of this type or Handler Type 3, then each such structure shall have a different instance ID value.</p>
9 [24:16]	RESERVED	—	—	—
1 [15]	NTH_TICK_SUPPORTED	R	IMPL	<p><b>Every N-th Tick Supported</b>            Defines whether this Scheduled Commands handler supports command execution on every <i>N</i>-th opportunity.  <b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>1'b1:</b> Supported; field <b>N_TICKS</b> in register <b>SCHEDULE_TABLE_##_CONFIG</b> is writeable.</li> <li>• <b>1'b0:</b> Not supported; field <b>N_TICKS</b> in register <b>SCHEDULE_SEQ_CONFIG</b> is reserved.</li> </ul>
3 [14:12]	RESERVED	—	—	—

Size [Bits]	Field Name	Memory Access	Reset Value	Description
4 [11:8]	CMD_SEQ_LENGTH	R	IMPL	<p><b>Command Sequence Length</b>  Number of Command Descriptor registers that can be stored in this structure.</p> <p><b>Note:</b> The field is zero-based, and the number of Command Descriptors is <math>2 + (\text{the field value})</math>.</p> <p><b>Valid Values:</b> 4'd0 (2 Command Descriptors) – 4'd6 (8 Command Descriptors)  All other values are reserved.</p>
8 [7:0]	SEQ_BASE_OFFSET	R	IMPL	<p><b>Schedule Sequence Base Offset</b>  Base Offset of the Schedule Sequence registers, in DWORDs, relative to the offset of this register.  A value of 0x2 indicates that the registers start at relative offset 0x10 within the Extended Capability structure, and immediately follow the <b>SCHEDULE_CONFIG</b> register.</p> <p><b>Valid Values:</b> 2–255</p>

6475 This register is used to program the Schedule timing in terms of ticks (at the Schedule Clock resolution),  
6476 and to run or stop the execution of Scheduled Commands.

6477 **Table 106 Schedule Configuration (SCHEDULE\_CONFIG) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [31]	R/S	R/W	0x0	<p><b>Scheduled Command Run/Stop</b>          Enables or disables processing for this instance of Scheduled Commands logic. Any modification to the Schedule Sequence registers shall only be performed while in <b>STOPPED</b> state.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: <b>STOPPED</b>: The Host Controller shall not run the Scheduled Commands schedule</li> <li>• 1'b1: <b>ENABLED</b>: The Host Controller shall run the Scheduled Commands schedule</li> </ul>
11 [30:20]	RESERVED	-	-	-
20 [19:0]	TICK_TIME	R/W	-	<p><b>Tick Time</b>          Indicates the number of clock cycles that are used to define the tick interval (per <a href="#">Section 6.16.1</a>). If the value of this field is equal to the value of field <b>CLOCK_RESOLUTION</b> (in register <b>SCHEDULE_CAPABILITIES</b>, <a href="#">Section 7.7.9.1</a>), then the tick interval shall be 1 ms.</p> <p><b>Valid Values:</b> 0–1,048,575</p>

#### 7.7.9.4 Schedule Sequence Mask (SCHEDULE\_SEQ\_MASK) (+ offset P varies)

This register defines the bit mask to be applied (using bitwise logical AND) to the schedule tick interval number when determining when the Host Controller shall start iterating over its configured Target Devices and executing the Command Descriptor sequence for each Target.

Since the location and size of the Schedule Sequence registers within the Extended Capability structure can vary, the offset of this register might also vary. The following formula gives the offset in DWORDs within the Extended Capability structure, relative to the start of the structure:

$$\text{Offset} = (2 + \text{SEQ\_BASE\_OFFSET})$$

**Note:**

*Field SEQ\_BASE\_OFFSET is found in register SCHEDULE\_LAYOUT for this Extended capability structure (Section 7.7.9.2).*

*Subsequent registers are described with offsets relative to this register, where the offset of this register is listed as P.*

For the lowest possible value of field **SEQ\_BASE\_OFFSET** (i.e., 0x2), the first Schedule Sequence register will reside at relative offset 0x10 bytes from the start of the Extended Capability structure.

**Table 107 Schedule Sequence Mask (SCHEDULE\_SEQ\_MASK) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	SCHEDULE_MASK	R/W	–	<p><b>Schedule Information</b></p> <p>Bit mask to apply to the schedule tick interval number when determining when the associated Command is to be executed.</p> <p>The Host Controller shall process this register one bit at a time, starting from Bit[0].</p> <p><b>Valid Values:</b> 0x0–0xFFFFFFFF, limited by the number of ticks supported by the Scheduled Commands logic.</p>

### 7.7.9.5 Schedule Sequence Config (SCHEDULE\_SEQ\_CONFIG) (P + 0x4)

If the Host Controller supports Scheduled Commands on every  $N$ -th opportunity, then this register's field **N\_TICKS** defines the value of  $N$ . If  $N$  is greater than 1, then the Host Controller shall only process the schedule on the  $N$ -th iteration (i.e., shall treat the schedule as empty for the first  $(N - 1)$  tick horizons).

This register also defines which Target fields in the subsequent array **SCHEDULE\_SEQ\_TARGETS\_#** registers to treat as valid. If the mask bit corresponding to a given Target is set to 1'b1, then the Host Controller shall include that Target field in these subsequent registers.

Within the Extended Capabilities structure, this register immediately follows register **SCHEDULE\_SEQ\_MASK** (i.e., at offset P + 0x4). Since the location and size of the Schedule Sequence registers within the Extended Capability structure can vary, the offset of this register might also vary. The following formula gives the offset in DWORDs within the Extended Capability structure, relative to the start of the structure:

$$\text{Offset} = (3 + \text{SEQ\_BASE\_OFFSET})$$

**Note:**

*Field SEQ\_BASE\_OFFSET is found in register SCHEDULE\_LAYOUT for this Extended capability structure (Section 7.7.9.2).*

6508

**Table 108 Schedule Sequence Config (SCHEDULE\_SEQ\_CONFIG) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
<b>16</b> [31:16]	<b>TARGET_MASK</b>	R/W	0x0	<p><b>Target Valid Mask</b>            Bit mask indicating which Targets to treat as valid.</p> <p><b>Bits:</b>            Mask Bit[N] corresponds to Target (N – 16) across the subsequent array of 1 to 4 <b>SCHEDULE_SEQ_TARGETS_#</b> registers, with 4 Targets per register:</p> <ul style="list-style-type: none"> <li>• <b>Bits[19:16]:</b> Register <b>SCHEDULE_SEQ_TARGETS_00</b>, Targets 0–3</li> <li>• <b>Bits[23:20]:</b> Register <b>SCHEDULE_SEQ_TARGETS_01</b> (if present), Targets 4–7</li> <li>• <b>Bits[27:24]:</b> Register <b>SCHEDULE_SEQ_TARGETS_02</b> (if present), Targets 8–11</li> <li>• <b>Bits[31:28]:</b> Register <b>SCHEDULE_SEQ_TARGETS_03</b> (if present), Targets 12–15</li> </ul> <p><b>Bit Position Values:</b></p> <ul style="list-style-type: none"> <li>• <b>1'b0:</b> Corresponding Target-specific field is not valid; do not execute the associated sequence of Command Descriptors for the indicated Target.</li> <li>• <b>1'b1:</b> Corresponding Target-specific is valid; include the indicated Target when iterating the associated sequence.</li> </ul> <p><b>Note:</b>  <i>Bits for Target-specific registers that are not present shall be read-only, with a fixed value of 1'b0.</i></p>
<b>3</b> [15:13]	<b>IBI_RING_BUNDLE</b> <i>(Only used for DMA Mode)</i>	R/W	0x0	<p><b>IBI Ring Bundle</b>            This field is only used if the Host Controller is operating in DMA Mode. The value of this field indicates the Ring Bundle ID for the IBI Ring Pair that will be used for IBI Status Descriptors and optional IBI Data Chunks that are generated by executing Transfer Commands for each iteration.</p> <p><b>Valid Values: 0 – 7</b> (must be a valid Ring Bundle ID)</p>
<b>5</b> [12:8]	RESERVED	—	—	—

<b>8</b> [7:0]	<b>N_TICKS</b>	R/W	-	<b>Schedule Information</b>  The value programmed in this field defines <i>N</i> . This field is only valid if the HW supports scheduling the command on every <i>N</i> -th opportunity, and if field <b>NTH_TICK_SUPPORTED</b> in register <b>SCHEDULE_TABLE_LAYOUT</b> indicates that this register is implemented. <b>Valid Values:</b> 0–255 <b>Values:</b> <ul style="list-style-type: none"><li>• The value is 0-based. Examples:<ul style="list-style-type: none"><li>• A field value of 8'd0 (<i>N</i>=1) would mean the associated Command would execute at every opportunity</li><li>• A field value of 8'd4 (<i>N</i> =5) would mean the associated Command would execute at every 5<sup>th</sup> opportunity</li></ul></li></ul>
-------------------	----------------	-----	---	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### 7.7.9.6 Schedule Sequence Targets (**SCHEDULE\_SEQ\_TARGETS\_##**) (P + 0x8, etc.)

This register array defines which Target Devices will be included when the Host Controller iterates over the bit mask of valid Targets (i.e., those that are set to 1'b1 in field **TARGET\_MASK** in register **SCHEDULE\_SEQ\_CONFIG**, [Section 7.7.9.4](#)). Each such register has 4 fields, each of which holds either the Target's Address (if Transfer Command Format 2 is supported) or the Target's DAT index (if Transfer Command Format 1 is supported).

This array has a minimum size of 1 register and a maximum size of 4 registers. Implementers shall support at least one register (for up to 4 Targets) and may support up to four registers (for 16 Targets):

- Register **SCHEDULE\_SEQ\_TARGETS\_00** (required) shall contain fields for Targets 0–3.  
These Targets are enabled and disabled by Bits[19:16] in field **TARGET\_MASK**.
- Register **SCHEDULE\_SEQ\_TARGETS\_01** (optional) shall contain fields for Targets 4–7.  
These Targets are enabled and disabled by Bits[23:20] in field **TARGET\_MASK**.
- Register **SCHEDULE\_SEQ\_TARGETS\_02** (optional) shall contain fields for Targets 8–11.  
These Targets are enabled and disabled by Bits[27:24] in field **TARGET\_MASK**.
- Register **SCHEDULE\_SEQ\_TARGETS\_03** (optional) shall contain fields for Targets 12–15.  
These Targets are enabled and disabled by Bits[31:28] in field **TARGET\_MASK**.

Any **SCHEDULE\_SEQ\_TARGETS** registers that are not implemented shall be read-only, with zero values.

Within the Extended Capabilities structure, the first register in this array immediately follows register **SCHEDULE\_SEQ\_CONFIG** (i.e., offset P + 0x8). Since the location and size of the Schedule Sequence registers within the Extended Capability structure can vary, the offset of these registers might also vary. The following formula gives the offset in DWORDs within the Extended Capability structure, relative to the start of the structure:

$$\begin{aligned} \text{Offset of first register} &= (4 + \text{SEQ\_BASE\_OFFSET}) \\ \text{Offset of second register} &= (5 + \text{SEQ\_BASE\_OFFSET}) \\ \text{Offset of third register} &= (6 + \text{SEQ\_BASE\_OFFSET}) \\ \text{Offset of fourth register} &= (7 + \text{SEQ\_BASE\_OFFSET}) \\ \text{Etc.} & \end{aligned}$$

**Note:**

Field **SEQ\_BASE\_OFFSET** is found in register **SCHEDULE\_LAYOUT** for this Extended capability structure ([Section 7.7.9.2](#)).

Subsequent registers for the Command Descriptors will start at 4 DWORDs after the first register (i.e., starting at offset P + 0x18).

For the lowest possible value of field **SEQ\_BASE\_OFFSET** (i.e., 0x2), the first instance of this register (i.e., for array index 0) will reside at relative offset 0x14 bytes from the start of the Extended Capability structure. Each subsequent instance shall reside immediately after the previous instance.

**Note:**

In the title of [Table 109](#), the hash characters ("##") represent the index for the array of these registers.

6547

**Table 109 Schedule Sequence Targets ## (SCHEDULE\_SEQ\_TARGETS\_##) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [31]	RESERVED	-	-	-
7 [30:24]	TARGET_D <i>where D =</i> 3 for SCHEDULE_SEQ_TARGETS_00 7 for SCHEDULE_SEQ_TARGETS_01 11 for SCHEDULE_SEQ_TARGETS_02 15 for SCHEDULE_SEQ_TARGETS_03	R/W	-	<b>Target D Address or Index</b> If Transfer Command Format 1 is supported, then Bits[28:24] of this field contain the DAT index for Target D, and Bits[30:29] are reserved. If Transfer Command Format 2 is supported, then this field contains the Target Address for Target D. A value of 0x0 is considered invalid.
1 [23]	RESERVED	-	-	-
7 [22:16]	TARGET_C <i>where C =</i> 2 for SCHEDULE_SEQ_TARGETS_00 6 for SCHEDULE_SEQ_TARGETS_01 10 for SCHEDULE_SEQ_TARGETS_02 14 for SCHEDULE_SEQ_TARGETS_03	R/W	-	<b>Target C Address or Index</b> If Transfer Command Format 1 is supported, then Bits[20:16] of this field contain the DAT index for Target C, and Bits[22:21] are reserved. If Transfer Command Format 2 is supported, then this field contains the Target Address for Target C. A value of 0x0 is considered invalid.
1 [15]	RESERVED	-	-	-
7 [14:8]	TARGET_B <i>where B =</i> 1 for SCHEDULE_SEQ_TARGETS_00 5 for SCHEDULE_SEQ_TARGETS_01 9 for SCHEDULE_SEQ_TARGETS_02 13 for SCHEDULE_SEQ_TARGETS_03	R/W	-	<b>Target B Address or Index</b> If Transfer Command Format 1 is supported, then Bits[12:8] of this field contain the DAT index for Target B, and Bits[14:13] are reserved. If Transfer Command Format 2 is supported, then this field contains the Target Address for Target B. A value of 0x0 is considered invalid.
1 [7]	RESERVED	-	-	-
5 [6:0]	TARGET_A <i>where A = 0</i> 0 for SCHEDULE_SEQ_TARGETS_00 4 for SCHEDULE_SEQ_TARGETS_01 8 for SCHEDULE_SEQ_TARGETS_02 12 for SCHEDULE_SEQ_TARGETS_03	R/W	-	<b>Target A Address or Index</b> If Transfer Command Format 1 is supported, then Bits[4:0] of this field contain the DAT index for Target A, and Bits[6:5] are reserved. If Transfer Command Format 2 is supported, then this field contains the Target Address for Target A. A value of 0x0 is considered invalid.

### 7.7.9.7 Schedule Sequence Command Descriptor (SCHEDULE\_SEQ\_COMMAND\_##) (P + 0x18, etc.)

The Extended Capability structure shall include registers containing the Transfer Commands to be executed at the scheduled time(s). The number of registers shall be sufficient to store between 2 and 8 complete Command Descriptors (as defined by field **CMD\_SEQ\_LENGTH**), where each Command Descriptor contains a Transfer Command in the format supported by the Host Controller, as indicated by field **CMD\_SIZE** in the **HC\_CAPABILITIES** register (see *Section 7.4.4*).

Since the location and size of the Schedule Sequence Entries within the Extended Capability structure can vary, the offset of each register in this array might also vary. Within the Extended Capabilities structure, the first register in this array immediately follows the array of four **SCHEDULE\_SEQ\_TARGETS** registers, (i.e., starting at offset P + 0x18).

The following formulas give the offset in DWORDs within the Extended Capability structure relative to the array index for the entry, where the index can vary from 0 to **CMD\_SEQ\_LENGTH**:

Offset of first Command Descriptor, first DWORD	= (8 + SEQ_BASE_OFFSET)
Offset of first Command Descriptor, second DWORD	= (9 + SEQ_BASE_OFFSET)
Offset of second Command Descriptor, first DWORD	= (10 + SEQ_BASE_OFFSET)
Offset of second Command Descriptor, second DWORD	= (11 + SEQ_BASE_OFFSET)
Etc.	

**Note:**

Fields **CMD\_SEQ\_LENGTH** and **SEQ\_BASE\_OFFSET** are found in register **SCHEDULE\_LAYOUT** for this Schedule Sequence (*Section 7.7.9.2*).

For the lowest possible value of field **SEQ\_BASE\_OFFSET** (i.e., 0x2), the first register for the first Command Descriptor in this sequence will reside at relative offset 0x28 bytes from the start of the Extended Capability structure.

The Command Descriptor is defined in *Section 8.4*.

For each set of registers that holds a Command Descriptor in the array, the Driver shall write either:

- A Command Descriptor containing a valid Transfer Command as defined in *Section 6.16.3*:

The value of field **CMD\_ATTR** shall be valid (i.e., shall indicate either a Regular Data Transfer Command or an Immediate Data Transfer Command).

The Transfer Command shall have either:

- **For Broadcast CCCs:** A value of 0x0 in the field that indicates the Target (since the Host Controller ignores the Target field for Broadcast CCCs); or
- **For any other transfer types:** A value of 0x1 in the field that indicates the Target, i.e., either the DAT index or the Target Address, depending on the supported Transfer Command Format (since the Host Controller will substitute the Target based on the specific iteration that it is currently executing).

Or:

- An invalid Command Descriptor to signal the end of the sequence.

The value 0x7 shall be used in field **CMD\_ATTR** to inform the Host Controller that the sequence of Transfer Commands has ended, so that the Host Controller will stop execution (i.e., will not process this Command Descriptor).

**Note:**

The Scheduled Commands logic does not support Command Descriptors with the Command type "Internal Control".

For each iteration of the Transfer Commands sequence, the Host Controller shall begin by executing the first Command Descriptor in this array with the indicated Target, then continue executing the next Command Descriptor in the sequence, until it either runs out of Command Descriptors, or encounters an

6592 error caused by a failed Transfer Command, or sees the value of 0x7 in field **CMD\_ATTR** which signals the  
6593 end of the sequence.

6594 After executing each Transfer Command in the sequence, the Host Controller shall generate an IBI Status  
6595 Descriptor (see *Section 8.6.5*), followed by IBI Data Chunks as appropriate for the particular Transfer  
6596 Command to indicate the status of executing the Transfer Command.

- 6597 • Field **STATUS\_TYPE** shall be set to 3'b010.
- 6598 • Field **ERROR** shall be set to 1'b0 if the Transfer Command completed successfully, or 1'b1 if there was an  
6599 error.

6600 For an error, no subsequent Transfer Commands in the sequence shall be executed for this Target.  
6601 The Host Controller may then either iterate to the next valid Target (if indicated), or else stop  
6602 execution entirely and report an interrupt to the Host via field **HC\_INTERNAL\_ERR\_STAT**.

- 6603 • Field **IBI\_ID** shall be set to a value that indicates which Command Descriptor in the array was executed,  
6604 and shall also indicate that the Transfer Command was executed by Scheduled Commands logic with  
6605 Handler Type 2. Bits[11:8] shall indicate the array index, Bit[12] shall be set to 1'b1, and Bits[15:13]  
6606 shall contain the 3-bit instance identifier, field **HT2\_INST\_ID**.

### 7.7.10 Scheduled Commands for Handler Type 3 (ID = 0x0F)

This Section defines the Capability-specific registers available under Extended Capability ID 0x0F, Scheduled Commands for Handler Type 3 (Schedule Buffer). The Scheduled Commands capability is an optional capability that allows the Host Controller to issue certain Transfer Commands on a programmed schedule (see [Section 6.16](#)) while operating in Active Controller mode. The Host Controller uses registers to describe and point to Host system memory that contains Transfer Descriptors and other configuration parameters relating to this instance of Scheduled Commands capability and its logic within the Host Controller.

**Note:**

*This Extended Capability structure only applies to Handler Type 3 (Schedule Buffer), as defined in [Section 6.16.4](#).*

The data structure shall contain, in the following order:

- A **SCHEDULE\_CAPABILITIES** register ([Section 7.7.10.1](#)) that indicates the basic timing parameters and clock resolution that software can configure for unique Transfer Commands.
- A **SCHEDULE\_LAYOUT** register ([Section 7.7.10.2](#)) that defines this structure for Handler Type 3, and describes the layout of additional registers.
- A **SCHEDULE\_CONFIG** register ([Section 7.7.10.3](#)) that contains run-time control over the operation of this instance of Scheduled Commands logic.

These first three registers are required, and shall serve as a descriptor for the Extended Capability structure, residing at a fixed offset relative to the starting offset for the structure (i.e., the instance of **EXTCAP\_HEADER**).

After these, the data structure shall contain additional registers that configure, point to, and describe the Schedule Buffer:

- Register **SCHEDULE\_BUF\_MASK** ([Section 7.7.10.4](#)) indicates the specific clock ticks that will activate the execution of the command sequence in the Schedule Buffer.
- Register **SCHEDULE\_BUF\_CONFIG** ([Section 7.7.10.5](#)) indicates how often the Transfer Commands in the Schedule Buffer shall be executed (i.e., only execute on every **N**-th cycle of ticks, if supported) and also where the Host Controller will return read data from Transfer Commands.
- If the Host Controller is in DMA Mode, then this register also holds the Ring Bundle ID for the IBI Ring Pair to be used for any IBI Status Descriptors and optional IBI Data Chunks that will be generated as a result of processing Transfer Commands.
- Registers **SCHEDULE\_BUF\_DESCR** ([Section 7.7.10.6](#)), **SCHEDULE\_BUF\_BASE\_LO** ([Section 7.7.10.7](#)), and **SCHEDULE\_BUF\_BASE\_HI** ([Section 7.7.10.8](#)) describe and point to the Host system memory that contains the Schedule Buffer (i.e., an array of Transfer Descriptors).
  - Note that a Host Controller that supports Handler Type 3 also implicitly supports Scatter-Gather capabilities for the Host system memory that holds the Schedule Buffer (per [Section 6.2.1](#)).

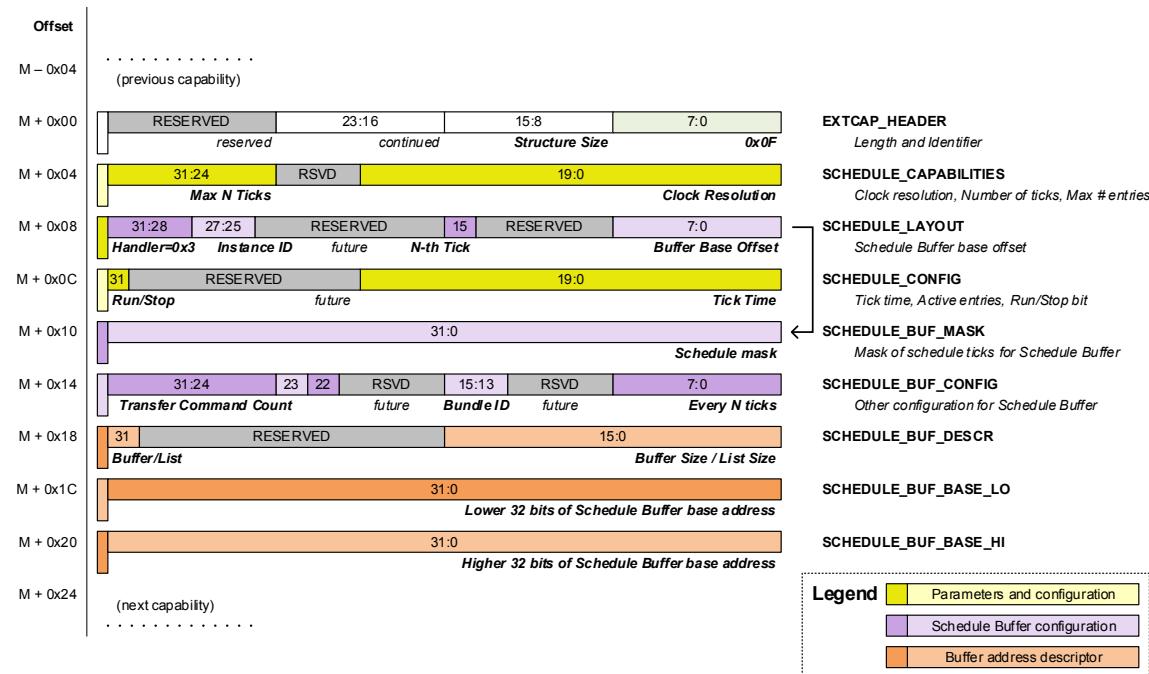
Field **SCHED\_HANDLER** in register **SCHEDULE\_LAYOUT** defines the overall format of the table and its registers.

**Note:**

*Subsequent text and register field tables assume a handler method value of 0x3, which defines this structure for a Schedule Buffer that software may populate with Transfer Descriptors.*

## 6647 Structure Template and Examples

6648 The following examples show how this type of Extended Capability structure could be defined, for several  
6649 different Host Controller implementations. For these examples, the structures are shown with a starting  
6650 offset **M** within the register offset map. It is assumed that this offset is within the range that register  
6651 **EXT\_CAPS\_SECTION\_OFFSET** (*Section 7.7*) refers to, and that it can be successfully read by the Driver.



6652 **Figure 48 Scheduled Commands Structure Example (Handler Type 3)**

6653 The **Figure 48** example shows a Host Controller that supports Scheduled Commands with a Schedule  
6654 Buffer that can contain Transfer Descriptors for any valid Targets.

6655 Additional implementation choices are possible, including the following configuration options:

- 6656 • The Schedule Buffer registers might start at a higher byte offset than 0x10, indicated by different values  
6657 of field **BUF\_BASE\_OFFSET** (i.e., values greater than 0x2), thereby leaving space for additional  
6658 configuration registers that might be present after register **SCHEDULE\_CONFIG** and before register  
6659 **SCHEDULE\_BUF\_MASK**.

### 7.7.10.1 Schedule Capabilities (SCHEDULE\_CAPABILITIES) (+ 0x04)

This register is used to provide information on the Host Controller's capabilities relating to the Scheduled Commands logic, including clock resolution and the maximum number of Commands that can be scheduled.

**Table 110 Schedule Capabilities (SCHEDULE\_CAPABILITIES) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
8 [31:24]	<b>MAX_N_TICKS</b>	R	IMPL	<p><b>Max N Ticks</b>            Hardware may implement a mechanism to support execution on every <math>N</math>-th opportunity. This value defines the maximum supported value for <math>N</math>.</p> <p><b>Valid Values:</b> 0–255  <b>Values:</b></p> <ul style="list-style-type: none"> <li>The value is 0-based.</li> </ul> <p><b>Examples:</b></p> <ul style="list-style-type: none"> <li>A field value of 8'd1 would represent 2 ticks (i.e., <math>1 \leq N \leq 2</math>)</li> <li>A field value of 8'd4 would represent 5 ticks (i.e., <math>1 \leq N \leq 5</math>)</li> <li><b>0:</b> Execution every <math>N</math>-th opportunity is not supported.</li> </ul>
4 [23:20]	RESERVED	—	—	—
20 [19:0]	<b>CLOCK_RESOLUTION</b>	R	IMPL	<p><b>Schedule Clock Resolution</b>            Number of clock ticks per 1 ms.  <b>Valid Values:</b> 0–1,048,575</p>

### 7.7.10.2 Schedule Layout (SCHEDULE\_LAYOUT) (+ 0x08)

This register defines the starting offset of the Schedule Buffer registers within the Extended Capability structure. This register also allows for this structure to be extensible for future variants (i.e., for other Handler Types or for other implementations of Scheduled Commands logic).

**Table 111 Schedule Layout (SCHEDULE\_LAYOUT) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
4 [31:28]	SCHED_HANDLER	R	IMPL	<p><b>Schedule Handler</b> Defines the layout of subsequent registers (i.e., relative offset 0x10 and above). <b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>0x3:</b> Single Schedule Buffer that contains Transfer Descriptors</li> <li>• All other values are either defined for other Handler Types, or reserved for future use</li> </ul>
3 [27:25]	HT3_INST_ID	R	IMPL	<p><b>Instance ID for Handler Type 3</b> This shall be a unique value between 3'd0 and 3'd7. If the Host Controller contains multiple Extended Capability structures of this type or Handler Type 2, then each such structure shall have a different instance ID value.</p>
9 [24:16]	RESERVED	—	—	—
1 [15]	NTH_TICK_SUPPORTED	R	IMPL	<p><b>Every N-th Tick Supported</b> Defines whether this Scheduled Commands handler supports command execution on every <i>N</i>-th opportunity. <b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>1'b1:</b> Supported; field <b>N_TICKS</b> in register <b>SCHEDULE_BUF_CONFIG</b> is writeable.</li> <li>• <b>1'b0:</b> Not supported; field <b>N_TICKS</b> in register <b>SCHEDULE_BUF_CONFIG</b> is reserved.</li> </ul>
7 [14:8]	RESERVED	—	—	—

Size [Bits]	Field Name	Memory Access	Reset Value	Description
8 [7:0]	<b>BUF_BASE_OFFSET</b>	R	IMPL	<b>Schedule Buffer Base Offset</b> Base Offset of the Schedule Buffer registers, in DWORDs, relative to the offset of this register. A value of 0x2 indicates that the registers start at relative offset 0x10 within the Extended Capability structure, and immediately follow the <b>SCHEDULE_CONFIG</b> register. <b>Valid Values:</b> 2–255

### 7.7.10.3 Schedule Configuration (SCHEDULE\_CONFIG) (+ 0x0C)

6668 This register is used to program the Schedule timing in terms of ticks (at the Schedule Clock resolution),  
6669 and to run or stop the execution of Scheduled Commands.

6670 **Table 112 Schedule Configuration (SCHEDULE\_CONFIG) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [31]	R/S	R/W	0x0	<p><b>Scheduled Command Run/Stop</b>          Enables or disables processing for this instance of Scheduled Commands logic. Any modification to the Schedule Buffer registers or the contents of the Schedule Buffer (i.e., in Host system memory) shall only be performed while in <b>STOPPED</b> state, unless otherwise noted (i.e., for specific fields).</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: <b>STOPPED</b>: The Host Controller shall not run the Scheduled Commands buffer</li> <li>• 1'b1: <b>ENABLED</b>: The Host Controller shall run the Scheduled Commands buffer (unless it is prevented from doing so, per field <b>EXECUTED</b>)</li> </ul>
11 [30:20]	RESERVED	-	-	-
20 [19:0]	TICK_TIME	R/W	-	<p><b>Tick Time</b>          Indicates the number of clock cycles that are used to define the tick interval (per <b>Section 6.16.1</b>).          If the value of this field is equal to the value of field <b>CLOCK_RESOLUTION</b> (in register <b>SCHEDULE_CAPABILITIES</b>, <b>Section 7.7.10.1</b>) then the tick interval shall be 1 ms.</p> <p><b>Valid Values:</b> 0–1,048,575</p>

#### 7.7.10.4 Schedule Buffer Mask (SCHEDULE\_BUF\_MASK) (+ offset varies)

This register defines the bit mask to be applied (using bitwise logical AND) to the schedule tick interval number when determining when the Host Controller shall start executing the Transfer Descriptors within the Schedule Buffer.

Since the location of the Schedule Buffer registers within the Extended Capability structure can vary, the offset of this register might also vary. The following formula gives the offset in DWORDs within the Extended Capability structure, relative to the start of the structure:

$$\text{Offset} = (2 + \text{BUF\_BASE\_OFFSET})$$

**Note:**

*Field **BUF\_BASE\_OFFSET** is found in register **SCHEDULE\_LAYOUT** for this Extended Capability structure (Section 7.7.10.2).*

*Subsequent registers are described with offsets relative to this register, where the offset of this register is listed as **P**.*

For the lowest possible value of field **BUF\_BASE\_OFFSET** (i.e., 0x2), this will reside at relative offset 0x10 bytes from the start of the Extended Capability structure.

**Table 113 Schedule Buffer Mask (SCHEDULE\_BUF\_MASK) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	<b>SCHEDULE_MASK</b>	R/W	–	<p><b>Schedule Information</b></p> <p>Bit mask to apply to the schedule tick interval number when determining when the associated Command is to be executed.</p> <p>The Host Controller shall process this register one bit at a time, starting from Bit[0].</p> <p><b>Valid Values:</b> 0x0–0xFFFFFFFF, limited by the number of ticks supported by the Scheduled Commands logic.</p>

### 7.7.10.5 Schedule Buffer Config (SCHEDULE\_BUF\_CONFIG) (P + 0x4)

If the Host Controller supports Scheduled Commands on every  $N$ -th opportunity, then this register's field **N\_TICKS** defines the value of  $N$ . If  $N$  is greater than 1, then the Host Controller shall only process the schedule on the  $N$ -th iteration (i.e., shall treat the schedule as empty for the first  $(N - 1)$  tick horizons).

This register also configures the method that the Host Controller will use to send read data from Read-Type transfers to the Host. This may be either IBI Status Descriptors with IBI Data Chunks (i.e., the same as Handler Types 1 and 2), or else the Data Buffer Descriptor within each Transfer Descriptor (i.e., akin to typical DMA Mode processing of Transfer Descriptors in a Command/Response Ring Pair).

Within the Extended Capabilities structure, this register immediately follows register **SCHEDULE\_BUF\_MASK** (i.e., at offset P + 0x4). Since the location of the Schedule Buffer registers within the Extended Capability structure can vary, the offset of this register might also vary. The following formula gives the offset in DWORDs within the Extended Capability structure, relative to the start of the structure:

$$\text{Offset} = (3 + \text{BUF\_BASE\_OFFSET})$$

**Note:**

Field **BUF\_BASE\_OFFSET** is found in register **SCHEDULE\_LAYOUT** for this Extended Capability structure ([Section 7.7.10.2](#)).

**Table 114 Schedule Buffer Config (SCHEDULE\_BUF\_CONFIG) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
8 [31:24]	<b>TRANSFER_COUNT</b>	R/O (volatile)	0x00	<p><b>Transfer Count</b>  This read-only field indicates the number of Transfer Descriptors that were processed and executed from the Schedule Buffer.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>0x00:</b> No Transfer Descriptors were executed (i.e., the Scheduled Commands Logic has not yet seen an opportunity to execute)</li> </ul> <p><b>Note:</b></p> <ul style="list-style-type: none"> <li>• When the Host clears field <b>EXECUTED</b>, this field shall automatically be reset to 0x00.</li> <li>• If field <b>RESPONSE_STEERING</b> is set to 1'b0, then this field always contains 0x00.</li> </ul> <ul style="list-style-type: none"> <li>• <b>0x01 – 0xFF:</b> Number of Transfer Descriptors that were executed</li> </ul>

1 [23]	<b>RESPONSE_STEERING</b>	R/W	0x0	<p><b>Response Steering</b>  Determines where read data for Read-Type transfers is received.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>1'b0:</b> Read data is received and sent to the Host in IBI Status Descriptor(s) with associated IBI Data Chunks.</li> <li>• For DMA Mode, these are enqueued to the IBI Ring Pair indicated by field <b>IBI_RING_BUNDLE</b>.</li> <li>• For PIO Mode, these are enqueued into the IBI Queue.</li> <li>• <b>1'b1:</b> Read data is received and sent to the Host in system memory that is addressed by the Data Buffer Descriptor (i.e., for each Transfer Descriptor).</li> </ul> <p><b>Note:</b>  <i>If this field is set to 1'b1, then the Host must pre-allocate sufficient system memory for each Data Buffer Descriptor that will receive data for its Read-Type transfer.</i></p>
1 [22]	<b>EXECUTED</b> <i>(Only used if read data will be stored in Data Buffer Descriptors)</i> <i>(May be cleared regardless of the value of field R/S)</i>	R/W1C	0x0	<p><b>Buffer Executed</b>  If field <b>RESPONSE_STEERING</b> is set to 1'b1, then the Host Controller shall set this field to 1'b1 after executing the Transfer Descriptors in the Schedule Buffer. This informs the Host that it must consume any read data, and that it may need to prepare the Data Buffer Descriptors for another iteration.  While this field is set to 1'b1, the Schedule Buffer will not be executed at the next opportunity. The Host must write 1'b1 to clear this field and resume execution at the next opportunity.</p>
6 [21:16]	RESERVED	—	—	—

<b>3</b> [15:13]	<b>IBI_RING_BUNDLE</b> <i>(Only used for DMA Mode)</i>	R/W	0x0	<b>IBI Ring Bundle</b> This field is only used if the Host Controller is operating in DMA Mode. The value of this field indicates the Ring Bundle ID for the IBI Ring Pair that will be used for IBI Status Descriptors and optional IBI Data Chunks that are generated by executing Transfer Commands from the Schedule Buffer. <b>Valid Values:</b> 0 – 7 (must be a valid Ring Bundle ID)
<b>5</b> [12:8]	RESERVED	–	–	–
<b>8</b> [7:0]	<b>N_TICKS</b>	R/W	–	<b>Schedule Information</b> The value programmed in this field defines $N$ . This field is only valid if the HW supports scheduling the command on every $N$ -th opportunity, and if field <b>NTH_TICK_SUPPORTED</b> in register <b>SCHEDULE_TABLE_LAYOUT</b> indicates that this register is implemented. <b>Valid Values:</b> 0–255 <b>Values:</b> <ul style="list-style-type: none"> <li>The value is 0-based.</li> </ul> Examples: <ul style="list-style-type: none"> <li>A field value of 8'd0 (<math>N=1</math>) would mean the associated Command would execute at every opportunity</li> <li>A field value of 8'd4 (<math>N=5</math>) would mean the associated Command would execute at every 5<sup>th</sup> opportunity</li> </ul>

6703

### 7.7.10.6 Schedule Buffer Descriptor (SCHEDULE\_BUF\_DESCR) (P + 0x8)

This register defines the size of the Schedule Buffer, as system memory that the Host must allocate to contain the Transfer Descriptors. The system memory may be either a single contiguous buffer, or a Scatter/Gather list of smaller buffers (per *Section 6.2.1*).

**Note:**

*This register and the subsequent SCHEDULE\_BUF\_BASE\_LO / SCHEDULE\_BUF\_BASE\_HI registers are largely similar to the contents of the Data Buffer Descriptor structure (see Section 8.3.1), and serve a similar purpose.*

Within the Extended Capabilities structure, this register immediately follows register SCHEDULE\_BUF\_CONFIG (i.e., at offset P + 0x8). Since the location of the Schedule Buffer registers within the Extended Capability structure can vary, the offset of this register might also vary. The following formula gives the offset in DWORDs within the Extended Capability structure, relative to the start of the structure:

$$\text{Offset} = (4 + \text{BUF\_BASE\_OFFSET})$$

**Note:**

*Field BUF\_BASE\_OFFSET is found in register SCHEDULE\_LAYOUT for this Extended Capability structure (Section 7.7.10.2).*

For the lowest possible value of field SEQ\_BASE\_OFFSET (i.e., 0x2), this register will reside at relative offset 0x14 bytes from the start of the Extended Capability structure.

**Table 115 Schedule Buffer Descriptor (SCHEDULE\_BUF\_DESCR) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [31]	BLP	R/W	0x0	<p><b>Buffer vs. List Pointer</b>            Indicates whether the Schedule Buffer Pointer (i.e., fields BASE_LO / BASE_HI in registers SCHEDULE_BUF_BASE_LO / SCHEDULE_BUF_BASE_HI) points directly to the Schedule Buffer, or to a List of buffers describing the Schedule Buffer.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: <b>BUFFER</b>: the Schedule Buffer Pointer points directly to physical memory used as the Schedule Buffer</li> <li>• 1'b1: <b>LIST</b>: The Schedule Buffer Pointer points to physical memory comprising an array of Memory Descriptors (i.e., a Scatter-Gather List that might not necessarily point to physically contiguous memory)</li> </ul>
15 [30:16]	RESERVED	-	-	-

Size [Bits]	Field Name	Memory Access	Reset Value	Description
16 [15:0]	BLOCK_SIZE or LIST_SIZE	R/W	0x0	<p><b>Schedule Buffer Block Size or List Size</b></p> <ul style="list-style-type: none"> <li><b>If field BLP contains 1'b0:</b> Then this field contains the number of Bytes in the Schedule Buffer Block</li> <li><b>If field BLP contains 1'b1:</b> Then this field contains the number of Entries in the Scatter-Gather List</li> </ul> <p>When a Scatter-Gather List is used, all but the last list entry's memory block must be sized to a multiple of DWORD i.e., the last entry's memory block can be a non-multiple). The Host Controller shall pad, to align to a DWORD boundary.</p> <p><b>Note:</b> <i>If field BLP=1'b0, then this field should be set to a non-zero value.</i></p>

### 7.7.10.7 Schedule Buffer Base Address Low (SCHEDULE\_BUF\_BASE\_LO) (P + 0xC)

In combination with register Schedule Buffer Base Address High, this register indicates the location of the Schedule Buffer.

Within the Extended Capabilities structure, this register immediately follows register **SCHEDULE\_BUF\_DESCR** (i.e., at offset P + 0xC). Since the location of the Schedule Buffer registers within the Extended Capability structure can vary, the offset of this register might also vary. The following formula gives the offset in DWORDs within the Extended Capability structure, relative to the start of the structure:

$$\text{Offset} = (5 + \text{BUF\_BASE\_OFFSET})$$

**Note:**

*Field BUF\_BASE\_OFFSET is found in register SCHEDULE\_LAYOUT for this Extended Capability structure (Section 7.7.10.2).*

**Table 116 Schedule Buffer Base Address Low (SCHEDULE\_BUF\_BASE\_LO) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	BASE_LO	R/W	0x0	<p><b>Schedule Buffer Base Address Low</b></p> <p>Lower 32 bits of pointer to physical memory allocated for the Schedule Buffer.</p> <p>The Schedule Buffer is DWORD aligned, so the last two Address bits will always be 2'b00.</p>

#### 7.7.10.8 Schedule Buffer Base Address High (SCHEDULE\_BUF\_BASE\_HI) (P + 0x10)

In combination with register Schedule Buffer Base Address Low, this register indicates the location of the Schedule Buffer.

Within the Extended Capabilities structure, this register immediately follows register **SCHEDULE\_BUF\_BASE\_LO** (i.e., at offset P + 0x10). Since the location of the Schedule Buffer registers within the Extended Capability structure can vary, the offset of this register might also vary. The following formula gives the offset in DWORDs within the Extended Capability structure, relative to the start of the structure:

$$\text{Offset} = (6 + \text{BUF\_BASE\_OFFSET})$$

**Note:**

Field **BUF\_BASE\_OFFSET** is found in register **SCHEDULE\_LAYOUT** for this Extended Capability structure ([Section 7.7.10.2](#)).

**Table 117 Schedule Buffer Base Address High (SCHEDULE\_BUF\_BASE\_HI) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	BASE_HI	R/W	0x0	<b>Schedule Buffer Base Address High</b> Upper 32 bits of pointer to physical memory allocated for Schedule Buffer.

### 7.7.10.9 Transfer Descriptors in Schedule Buffer

The Schedule Buffer contains one or more Transfer Descriptors to be executed at the scheduled time. The Host must allocate a region of system memory that is large enough for the use case. The Schedule Buffer can hold up to 255 Transfer Descriptors, where each Transfer Descriptor contains a Transfer Command in the format supported by the Host Controller, as indicated by field **CMD\_SIZE** in the **HC\_CAPABILITIES** register (see [Section 7.4.4](#)).

The Command Descriptor is defined in [Section 8.4](#) and the Transfer Descriptor is defined in [Section 8.3](#).

**Note:**

*The size of the Transfer Descriptor used by Handler Type 3 is the same as the Transfer Descriptor size for supported Ring Bundles, as defined by field **XFER\_STRUCT\_SIZE** in register **CR\_SETUP** ([Section 7.6.10.1](#)). DWORDs 0 and 1 contain the Command Descriptor, DWORD 2 contains the Data Buffer Descriptor fields, and DWORDS 3 and 4 contain the Data Buffer Pointer.*

For each Transfer Descriptor that holds a Command Descriptor in the array, the Driver shall provide either:

- A Command Descriptor containing a valid Transfer Command as defined in [Section 6.16.3](#):

The value of field **CMD\_ATTR** shall be valid (i.e., shall indicate either a Regular Data Transfer Command or an Immediate Data Transfer Command).

The Transfer Command shall have either:

- **For Broadcast CCCs:** A value of 0x0 in the field that indicates the Target (since the Host Controller ignores the Target field for Broadcast CCCs); or
- **For any other transfer types:** A value of 0x1 in the field that indicates the Target, i.e., either the DAT index or the Target Address, depending on the supported Transfer Command Format (since the Host Controller will substitute the Target based on the specific iteration that it is currently executing).

Or:

- An invalid Command Descriptor to signal the end of the Schedule Buffer:

The value 0x7 shall be used in field **CMD\_ATTR** to inform the Host Controller that the valid Transfer Descriptors have ended, so that the Host Controller will stop execution (i.e., will not process this Transfer Descriptor).

**Note:**

*The Scheduled Commands logic does not support Command Descriptors with the Command type “Internal Control”.*

At each opportunity to start execution, the Host Controller shall begin by processing the first Transfer Descriptor in this Schedule Buffer, then continue processing the next Transfer Descriptors, until it either runs out of valid Transfer Descriptors in the Schedule Buffer, or encounters an error caused by a failed Transfer Command, or sees the value of 0x7 in field **CMD\_ATTR** which signals the end of valid Transfer Descriptors in the Schedule Buffer.

**Note:**

*The Host Controller shall not execute the Schedule Buffer while field **RESPONSE\_STEERING** and field **EXECUTED** (in register **SCHEDULE\_BUF\_CONFIG**) both contain 1'b1. Such values indicate that a previous execution of the Schedule Buffer generated read data, but the Host has not yet consumed it from the Data Buffer Descriptors. The Host must first consume this read data, and then clear field **EXECUTED**, before the Host Controller will resume execution of the Schedule Buffer.*

After executing each Transfer Command in the Schedule Buffer, the Host Controller shall generate an IBI Status Descriptor (see [Section 8.6.5](#)) followed by optional IBI Data Chunks with read data (as appropriate for the Transfer Command) to indicate the status of executing the Transfer Command, but only if field **RESPONSE\_STEERING** has a value of 1'b0. If, on the other hand, the value of field **RESPONSE\_STEERING** is 1'b1, then the read data shall instead be written into the Data Buffer Descriptor for its Transfer Descriptor.

Within each IBI Status Descriptor:

- 6793 • Field **STATUS\_TYPE** shall be set to 3'b010.  
6794 • Field **ERROR** shall be set to 1'b0 if the Transfer Command completed successfully, or 1'b1 if there was an  
6795 error.

6796 For an error, no subsequent Transfer Commands in the Schedule Buffer shall be executed. The Host  
6797 Controller shall stop execution entirely and report an interrupt to the Host via field  
6798 **HC\_INTERNAL\_ERR\_STAT**.

- 6799 • Field **IBI\_ID** shall be set to a value that indicates that one or more Transfer Commands were executed by  
6800 Scheduled Commands logic with Handler Type 3. Bit[12] shall be set to 1'b1, and Bits[15:13] shall  
6801 contain the 3-bit instance identifier, field **HT3\_INST\_ID**.

6802 **Note:**

6803 Since the Scheduled Commands logic does not know the actual number of valid Transfer  
6804 Descriptors in the Schedule Buffer when it starts execution, it is possible for many IBI Status  
6805 Descriptors (and optional IBI Data Chunks) to be generated, which the Host must consume in a  
6806 timely manner. In situations when the Host does not consume these from either the IBI Port (in PIO  
6807 Mode) or the configured IBI Ring Pair (in DMA Mode), and the Queue or Ring becomes full, the  
6808 Host Controller must temporarily stall execution of Transfer Descriptors in the Schedule Buffer until  
6809 the Host resolves the situation by consuming IBI Status Descriptors (and optional IBI Data Chunks,  
6810 depending on the value of field **RESPONSE\_STEERING**). If the stall becomes a timeout, then the  
6811 Host Controller shall stop processing and report the error. Such conditions will be reported to the  
6812 Host as interrupts; refer to fields **HC\_WARN\_CMD\_SEQ\_STALL\_STAT** and  
6813 **HC\_ERR\_CMD\_SEQ\_TIMEOUT\_STAT** in register **INTR\_STATUS** (see [Section 7.4.7](#)).

6814 At the end of execution, if field **RESPONSE\_STEERING** in register **SCHEDULE\_BUF\_CONFIG** has a value of  
6815 1'b1, then the Host Controller shall set field **EXECUTED** to 1'b1 and then prevent subsequent execution  
6816 attempts until the Host clears field **EXECUTED** at a later time.

### 7.7.11 Standby Controller Mode (ID = 0x12)

This Section defines the Capability-specific registers available under Extended Capability ID 0x12, Standby Controller mode. These registers define the capabilities of, and affect the configuration and control of, the Host Controller's Secondary Controller Logic. This logic is activated while the Host Controller is operating in Standby Controller mode, i.e., is in a Secondary Controller role.

For these registers, fields that the Host may change are indicated as either R/W or R/cW in the register field tables, and fields that are read-only to the Host are indicated as R. The intent and purpose of each field is described either in the register field tables, or in the subsections.

Some configuration fields (indicated as R/cW, meaning Read, conditional Write) are only writeable while the Host Controller's Secondary Controller Logic is inactive (i.e., disabled). R/cW fields are read-only from the Host while the Host Controller is operating in Standby Controller mode (i.e., while field **STBY\_CR\_ENABLE\_INIT** in register **STBY\_CR\_CONTROL** is set to a non-zero value; see [Section 7.7.11.1](#)).

Many such fields defined as conditionally writeable are used for initial configuration, and might be returned as response data on the I3C Bus for certain standard CCCs or during Dynamic Address Assignment, thereby providing information to the Active Controller of the I3C Bus. Since this data provided to the Active Controller would be treated as a contract, any changes desired by Host software might not be visible to the Active Controller, as the Active Controller would rely on the data being transferred during Bus Enumeration or any phases of initial configuration (e.g., Dynamic Address Assignment with [ENTDAA](#)). The Host Controller shall block writes to such fields indicated as R/cW from the Host while the Secondary Controller Logic is enabled, in order to prevent a violation of contract or other errors that might arise from mismatched expectations or deviations from previously agreed behavior. Such fields may only be written from the Host while the Secondary Controller Logic is disabled, and would typically require the Host Controller to re-join the I3C Bus.

Other register fields might be read-only to the Host, but are updated by the Secondary Controller Logic during the course of normal operation, as a result of changing conditions or new status based on I3C Bus activity (such as receiving certain CCCs with data messages or Defining Byte values). The register field tables in subsections below indicate these fields, along with any special considerations for implementers and software developers.

**Table 118 Register Map of Extended Capability Structure for Standby Controller Mode**

Offset	Offset + 0x0	Offset + 0x4	Offset + 0x8	Offset + 0xC
M + 0x00	<b>EXTCAP_HEADER</b>	<b>STBY_CR_CONTROL</b>	<b>STBY_CR_DEVICE_ADDR</b>	<b>STBY_CR_CAPABILITIES</b>
M + 0x10	-	<b>STBY_CR_STATUS</b>	<b>STBY_CR_DEVICE_CHAR</b>	<b>STBY_CR_DEVICE_PID_LO</b>
M + 0x20	<b>STBY_CR_INTR_STATUS</b>	-	<b>STBY_CR_INTR_SIGNAL_ENABLE</b>	<b>STBY_CR_INTR_FORCE</b>
M + 0x30	<b>STBY_CR_CCC_CONFIG_GETCAPS</b>	<b>STBY_CR_CCC_CONFIG_RSTACT_PARAMS</b>	-	-

6845 This register controls operation of a Host Controller that supports Standby Controller mode, including the configuration and control of its Secondary Controller Logic.  
6846

6847 **Table 119 Standby Controller Control (STBY\_CR\_CONTROL) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
2 [31:30]	STBY_CR_ENABLE_INIT	R/W	0x0	<p><b>Host Controller Secondary Controller Enable</b></p> <p>Enables or disables Secondary Controller operation on the I3C Bus by the Host Controller. Also controls initialization and Dynamic Address Assignment in Standby Controller mode (see <a href="#">Section 6.17.1</a>).</p> <p>This applies to the Secondary Controller Logic's autonomous responses to CCCs (see <a href="#">Section 6.17.3</a>).</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>2'b00: DISABLED:</b> Secondary Controller operation is disabled</li> <li>• <b>2'b01: ACM_INIT:</b> Secondary Controller operation is enabled, but Host Controller initializes in Active Controller mode</li> <li>• <b>2'b10: SCM_RUNNING:</b> Secondary Controller operation is enabled, Host Controller initializes in Standby Controller mode.</li> <li>• <b>2'b11: SCM_HOT_JOIN:</b> Secondary Controller operation is enabled, Host Controller conditionally becomes a Hot-Joining Device to receive its Dynamic Address before operating in Standby Controller mode.</li> </ul>
9 [29:21]	RESERVED	R	0x0	–

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [20]	RSTACT_DEFBYTE_02	R/W	0x0	<p><b>RSTACT Support DefByte 0x02</b>  Controls whether I3C Secondary Controller Logic supports (i.e., ACKs and responds to) <b>RSTACT</b> CCC with Defining Byte 0x02. This is used to inform the Driver when the Active Controller intends to perform a Full Chip Reset on this Host Controller, which the Driver should handle appropriately.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: <b>NOT_SUPPORTED</b>: Do not ACK Defining Byte 0x02</li> <li>• 1'b1: <b>HANDLE_INTR</b>: Support Defining Byte 0x02; handle with ACK and assert interrupt <b>STBY_CR_OP_RSTACT_STAT</b> (<b>Section 6.17.3.1.1</b>)</li> </ul>
4 [19:16]	RESERVED	R	0x0	–
1 [15]	DAA_ENTDAA_ENABLE	R/cW	0x0	<p><b>Dynamic Address Method Enable</b>  Indicates which Dynamic Address Assignment method(s) are enabled. Each enabled method must also be implemented, and must be supported by the corresponding field (i.e., Bits[15:13]) of register <b>STBY_CR_CAPABILITIES</b> (<b>Section 7.7.11.3</b>).  If the Host Controller is joining the I3C Bus in Standby Controller mode, then at least one supported mode must be enabled, in order to receive a Dynamic Address.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: <b>DISABLED</b>: will not respond</li> <li>• 1'b1: <b>ENABLED</b>: will respond</li> </ul>
1 [14]	DAA_SETDASA_ENABLE	R/cW	0x0	
1 [13]	DAA_SETAASA_ENABLE	R/cW	0x0	
1 [12]	TARGET_XACT_ENABLE	R/cW	0x1	<p><b>Target Transaction Interface Servicing Enable</b>  Indicates whether Read-Type and Write-Type transaction servicing is enabled, via an I3C Target Transaction Interface to software (<b>Section 6.17.3</b>).  This configuration may only be changed when the Secondary Controller Logic is in <b>DISABLED</b> state.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: <b>DISABLED</b>: not available</li> <li>• 1'b1: <b>ENABLED</b>: available for software</li> </ul>
1 [11]	RESERVED	R	0x0	–

Size [Bits]	Field Name	Memory Access	Reset Value	Description
3 [10:8]	<b>BCAST_CCC_IBI_RING</b>	R/W	0x0	<b>Ring Bundle IBI Selector for Broadcast CCC Capture</b> Indicates which Ring Bundle will be used to capture Broadcast CCC data sent by the Active Controller. Each such Broadcast CCC will be written to the IBI Ring Pair of this Ring Bundle (per <b>Section 6.17.3.2</b> ). The Ring Bundle must be configured and enabled, and its IBI Ring Pair must also be initialized and ready to receive data.
2 [7:6]	RESERVED	R	0x0	–
1 [5]	<b>CR_REQUEST_SEND</b>	W	0x0	<b>Send Controller Role Request</b> Recommended for Minimal Standby Controller implementations. If not supported, then writes shall have no effect. If supported, and whenever it is allowed, then a write of 1'b1 to this field shall instruct the Secondary Controller Logic to attempt to send a Controller Role Request to the I3C Bus, per <b>Section 6.17.3.3</b> .
1 [4]	<b>HANOFF_DEEP_SLEEP</b>	R/W (sticky)	0x0	<b>Handoff Deep Sleep</b> If this field has a value of 1'b1, then the Secondary Controller Logic shall report a return from Deep Sleep state to the Active Controller, per <b>Section 6.17.4.2</b> . Writing 1'b1 to this bit shall be 'sticky': software may set it, but cannot clear it (i.e., a write of 1'b0 shall have no effect). This field shall automatically clear to 1'b0 after accepting the Controller Role and transitioning to Active Controller mode.

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [3]	PRIME_ACCEPT_GETACCCR	R/W or R	0x1	<p><b>Prime to Accept Controller Role</b>  Determines whether the Host Controller shall prime the Secondary Controller Logic to automatically accept the Controller Role after a successful transition to Standby Controller mode.</p> <p>This value is latched when processing the special Command Descriptor to pass the Controller Role, per <b>Section 6.17.2.2</b>.</p> <p>For some minimal Host Controller implementations, this field might be read-only (i.e., might always be primed).</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>1'b0: DISABLED:</b> Do not prime the Secondary Controller Logic to automatically accept the Controller Role; software must therefore write 1'b1 to field <b>ACR_FSM_OP_SELECT</b></li> <li>• <b>1'b1: ENABLED:</b> Prime the Secondary Controller Logic to automatically accept the Controller Role after transitioning to Standby Controller mode</li> </ul>

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [2]	ACR_FSM_OP_SELECT	R/W or R/cW	0x1	<p><b>Active Controller Select</b>  Permits the Device to (1) Accept the Controller Role while operating in Standby Controller mode, and (2) Send the <b>GETACCCR</b> CCC while operating in Active Controller mode.  When read by software, this bit indicates the desired state for the Host Controller i.e., the “transition to” state). This bit should be interpreted in combination with field <b>AC_CURRENT_OWN</b> in register <b>PRESENT_STATE</b> which reflects the Controller Role (i.e., SCL Line driving status).</p> <p><b>Values for Active Controller mode:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: <b>HANOFF_READY_SEND</b>: The Host Controller is prepared to accept a Command Descriptor for the <b>GETACCCR</b> CCC from software. It shall be accepted as long as all pre-Handoff conditions remain true (see <b>Section 6.17.2</b>).</li> <li>• 1'b1: <b>BUS_OWNER</b>: The Host Controller retains the Controller Role and is not prepared to send a <b>GETACCCR</b> CCC.</li> </ul> <p><b>Values for Standby Controller mode:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: <b>NOT_BUS_OWNER</b>: The Host Controller is not in transition to the Active Controller state, and will not accept the Controller Role.</li> <li>• 1'b1: <b>HANOFF_READY_RECV</b>: The Host Controller is ready to accept the Controller Role via the <b>GETACCCR</b> CCC (see <b>Section 6.17.4</b>).</li> </ul>
1 [1]	HANOFF_DELAY_NACK	R/W1C	0x0	<p><b>Handoff Delay NACK</b>  Indicates whether the Device shall NACK any attempt to receive the Controller Role via <b>GETACCCR</b> CCC, due to being busy processing a <b>DEFTGTS</b> CCC or any <b>DEFGRPA</b> CCCs sent by the Active Controller.  The Secondary Controller Logic shall automatically set this bit to 1'b1 if it detects any incoming Broadcast <b>DEFTGTS</b> or <b>DEFGRPA</b> CCCs.  Software may clear this bit by writing the value 1'b1 (i.e., write 1 to clear) after processing these CCCs.  This bit is also used for automatic response to <b>GETSTATUS</b> Format 2 CCC with <b>PRECR</b> Defining Byte (see <b>Section 6.17.3.1.1</b>).</p>

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [0]	PENDING_RX_NACK	R/W1C	0x0	<p><b>Pending RX NACK</b></p> <p>Indicates whether the Device shall NACK any attempt to receive the Controller Role via <a href="#">GETACCR</a> CCC, due to pending Write-Type transfers not yet processed by software.</p> <p>The Secondary Controller Logic shall automatically set this bit to 1'b1 if it receives any incoming Write-Type transfers (per <a href="#">Section 6.17.4</a>) that are not Broadcast <a href="#">DEFTGTS</a> or <a href="#">DEFGRPA</a> CCCs. Software may clear this bit by writing the value to 1'b1 (i.e., write 1 to clear).</p>

### 7.7.11.2 Standby Controller Device Address (STBY\_CR\_DEVICE\_ADDR) (+ 0x08)

This register holds the Host Controller's Dynamic Address and Static Address while operating in Standby Controller mode. The Secondary Controller Logic uses these fields to match incoming I3C transactions, and during Dynamic Address Assignment (see *Section 6.17.1* and *Section 6.17.3*).

**Table 120 Standby Controller Device Address (STBY\_CR\_DEVICE\_ADDR) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [31]	DYNAMIC_ADDR_VALID	R	0x0	<p><b>Dynamic Address is Valid</b>            Indicates whether or not the value in the DYNAMIC_ADDR field is valid.            This bit is also visible from register <b>CONTROLLER_DEVICE_ADDR</b> (<i>Section 7.4.3</i>). It may be changed from that register, but only while the Controller is operating in Active Controller mode.            While operating in Standby Controller mode, the Host Controller shall set this bit to 1'b1 when the Primary Controller assigns the Dynamic Address (i.e., during the <b>ENTDAA</b> or <b>SETDASA/SETAASA</b> process).  <b>Values:</b> <ul style="list-style-type: none"> <li>• 1'b0: DYNAMIC_ADDR field is not valid</li> <li>• 1'b1: DYNAMIC_ADDR field is valid</li> </ul> </p>
8 [30:23]	RESERVED	R	0x0	–
7 [22:16]	DYNAMIC_ADDR	R	0x0	<p><b>Device Dynamic Address</b>            Contains the Host Controller Device's Dynamic Address. The Host Controller uses this I3C Address to respond to I3C Transactions as a Target or Secondary Controller.            This Dynamic Address is also visible from register <b>CONTROLLER_DEVICE_ADDR</b> (<i>Section 7.4.3</i>). It may be changed from that register, but only while the Host Controller is operating in Active Controller mode.            While operating in Standby Controller mode, the Host Controller sets this field to the Address assigned by the Primary Controller during Dynamic Address Assignment (i.e., during the <b>ENTDAA</b> or <b>SETDASA/SETAASA</b> procedure), or by subsequent use of the <b>SETNEWDA</b> CCC.</p>

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [15]	STATIC_ADDR_VALID	R/cW	0x0	<p><b>Static Address is Valid</b>            Indicates whether or not the value in the STATIC_ADDR field is valid.            While the Host Controller is configured for Standby Controller mode, software may set this bit to 1'b1 before enabling Secondary Controller Bus operation in register <b>STBY_CR_CONTROL</b> (i.e., before the <b>SETDASA/SETAASA</b> process).            This field is Read-Only while field <b>STBY_CR_ENABLE_INIT</b> in register <b>STBY_CR_CONTROL</b> (<b>Section 7.7.11.1</b>) is not zero.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: The Static Address field is not valid; the Host Controller may only participate in Dynamic Address Assignment with <b>ENTDAA</b> (if implemented and enabled).</li> <li>• 1'b1: The Static Address field is valid; the Host Controller may respond to <b>SETDASA</b> and/or <b>SETAASA</b> per the configuration in register <b>STBY_CR_CONTROL</b>.</li> </ul>
8 [14:7]	RESERVED	R	0x0	–
7 [6:0]	STATIC_ADDR	R/cW	0x0	<p><b>Device Static Address</b>            This field contains the Host Controller Device's Static Address. If field <b>STATIC_ADDR_VALID</b> is set to 1'b1, then the Host Controller shall use this Static Address when responding to <b>SETDASA</b> and/or <b>SETAASA</b>, if the relevant fields in register <b>STBY_CR_CONTROL</b> are set.            While the Controller is configured for Standby Controller mode, software may set this field to a valid I3C Address before enabling Secondary Controller Bus operation in register <b>STBY_CR_CONTROL</b> (<b>Section 7.7.11.1</b>).            This field is Read-Only while field <b>STBY_CR_ENABLE_INIT</b> in register <b>STBY_CR_CONTROL</b> (<b>Section 7.7.11.1</b>) is not zero.</p>

6852

**Note:**

6853

This register shall be used only when the Host Controller is operating in Standby Controller mode.  
 Register **CONTROLLER\_DEVICE\_ADDR** is intended for use in Active Controller mode.

6854

Use of a Static Address in Secondary Controller operation requires prior configuration on the Primary Controller of the I3C Bus. For most use cases, Dynamic Address Assignment with the **ENTDAA** CCC is the recommended option.

6855

6856

6857

### 7.7.11.3 Standby Controller Capabilities (STBY\_CR\_CAPABILITIES) (+ 0x0C)

This register defines the capabilities of a Host Controller that supports Standby Controller mode, including the capabilities of its Secondary Controller Logic, and what methods of Dynamic Address Assignment it supports while acting as a Secondary Controller. The Host Controller may also optionally support the I3C Target Transaction Interface to software (which is exposed separately as vendor-defined Extended Capabilities).

**Table 121 Standby Controller Capabilities (STBY\_CR\_CAPABILITIES) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
16 [31:16]	RESERVED	R	0x0	–
1 [15]	DAA_ENTDAA_SUPPORT <i>See Note below</i>	R	IMPL	Defines whether Dynamic Address Assignment with <a href="#">ENTDAA</a> CCC is supported. <b>Values:</b> <ul style="list-style-type: none"><li>• 1'b0: DISABLED: Not supported</li><li>• 1'b1: ENABLED: Supported</li></ul>
1 [14]	DAA_SETDASA_SUPPORT <i>See Note below</i>	R	IMPL	Defines whether Dynamic Address Assignment with <a href="#">SETDASA</a> CCC (using Static Address) is supported. <b>Values:</b> <ul style="list-style-type: none"><li>• 1'b0: DISABLED: Not supported</li><li>• 1'b1: ENABLED: Supported</li></ul>
1 [13]	DAA_SETAASA_SUPPORT <i>See Note below</i>	R	IMPL	Defines whether Dynamic Address Assignment with <a href="#">SETAASA</a> CCC (using Static Address) is supported. <b>Values:</b> <ul style="list-style-type: none"><li>• 1'b0: DISABLED: Not supported</li><li>• 1'b1: ENABLED: Supported</li></ul>
1 [12]	TARGET_XACT_SUPPORT	R	IMPL	Defines whether an I3C Target Transaction Interface is supported. If supported and enabled, then software may receive transactions and may optionally respond to Read-Type transactions (see <a href="#">Section 6.17.3</a> ). <b>Values:</b> <ul style="list-style-type: none"><li>• 1'b0: DISABLED: Not supported</li><li>• 1'b1: ENABLED: Supported via vendor-defined Extended Capability structure</li></ul>
6 [11:6]	RESERVED	R	0x0	–

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [5]	SIMPLE_CRR_SUPPORT	R	IMPL	<p>Defines whether the Host Controller supports the Simple Controller Role Request interface (see <b>Section 6.17.3.3</b>).</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: DISABLED: Not supported</li> <li>• 1'b1: ENABLED: Supported</li> </ul>
5 [4:0]	RESERVED	R	0x0	—

6864

**Note:**

6865

If the Host Controller will be joining the I3C Bus in the Secondary Controller role (per **Section 6.17.1.3**), then the Secondary Controller Logic shall support at least one of the Dynamic Address Assignment methods ([ENTDAA](#), [SETDASA](#), and/or [SETAASA](#)) and shall indicate this support in Bits[15:13] of this register. Such support might also be required if the Secondary Controller Logic will be used with the [RSTDAA](#) CCC and the Active Controller would re-assign the Dynamic Address after resetting all Dynamic Addresses on the I3C Bus. Additionally, Hot-Join Requests must be supported if Dynamic Address Assignment with [ENTDAA](#) is supported.

6866

Implementers should support the Controller Role Request capability in the Secondary Controller Logic, as this is the defined mechanism for a Secondary Controller to request the Controller Role from the Active Controller of the I3C Bus. Field **SIMPLE\_CRR\_SUPPORT** should have a value of 1'b1, except for special use cases where the Host Controller never sends a Controller Role Request in Standby Controller mode.

6867

6868

6869

6870

6871

6872

6873

6874

6875

6876

#### 7.7.11.4 Standby Controller Status (STBY\_CR\_STATUS) (+ 0x14)

6877

This register returns the current state of the Secondary Controller Logic.

6878

**Table 122 Standby Controller Status (STBY\_CR\_STATUS) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
23 [31:9]	RESERVED	R	0x0	–
1 [8]	HJ_REQ_STATUS	R	0x0	<p><b>Hot-Join Request Status</b>            Indicates the status of a pending or attempted Hot-Join Request, if the Driver had previously written 2'b11 to field <b>STBY_CR_ENABLE_INIT</b> in register <b>STBY_CR_CONTROL</b> to make such a request (see <a href="#">Section 6.17.1.3</a>).</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: No pending request. This is the default reset value.</li> <li>• 1'b1: The Host Controller successfully emitted a Hot-Join Request (i.e., at least once).</li> </ul>
3 [7:5]	SIMPLE_CRR_STATUS	R	0x0	<p><b>Simple Controller Role Request Status</b>            Indicates the status of a pending or attempted Controller Role Request, if the Driver had previously written to field <b>CR_REQUEST_SEND</b> in register <b>STBY_CR_CONTROL</b> to make such a request (see <a href="#">Section 6.17.3.3</a>).</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 3'b000: No pending request. This is the default reset value.</li> <li>• 3'b001: The request is pending, but has not yet been attempted due to Bus Available Condition.</li> <li>• 3'b010: The request was attempted, but was NACKed by the Active Controller.</li> <li>• 3'b011: The request was attempted, but the Active Controller NACKed it and sent the <b>DISEC</b> CCC with the <b>DISCR</b> bit set to disable further Controller Role Requests.</li> <li>• 3'b100: The request was attempted and the Active Controller ACKed it.</li> </ul> <p>All other values: Reserved for future use.</p>
2 [4:3]	RESERVED	R	0x0	–

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [2]	AC_CURRENT_OWN	R	0x0	<b>Active Controller</b> Mirrors the contents of field AC_CURRENT_OWN in register <b>PRESENT_STATE</b> (see <b>Section 7.4.6</b> ).
2 [1:0]	RESERVED	R	0x0	–

### 7.7.11.5 Standby Controller Device Characteristics (STBY\_CR\_DEVICE\_CHAR) (+ 0x18)

This register holds the Host Controller's I3C BCR and DCR while operating in Standby Controller mode. If the I3C PID is required (either by the [GETPID](#) CCC or by the [ENTDAA](#) procedure), then this register also holds Bits[15:0] of the I3C PID.

**Table 123 Standby Controller Device Characteristics (STBY\_CR\_DEVICE\_CHAR) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
3 [31:29]	BCR_FIXED	R	IMPL	<p><b>Bus Characteristics Register Fixed</b></p> <p>This field holds Bits[7:5] of the BCR for the Host Controller while operating in Standby Controller mode.</p> <p>This field is used to automatically respond to the <a href="#">GETBCR</a> CCC, and to the <a href="#">ENTDAA</a> procedure if supported.</p> <p>This field is Read-Only with a fixed value of 3'b011, since a Controller-capable Device compliant with I3C version 1.1.1+ is required to have specific values for BCR Bits[7:5].</p>
5 [28:24]	BCR_VAR	R/cW	0x0	<p><b>Bus Characteristics Register Variable</b></p> <p>This field holds Bits[4:0] of the BCR for the Host Controller while operating in Standby Controller mode.</p> <p>This field is used to automatically respond to the <a href="#">GETBCR</a> CCC, and to the <a href="#">ENTDAA</a> procedure if supported.</p> <p>This field is Read-Only while the <a href="#">STBY_CR_ENABLE_INIT</a> field in register <a href="#">STBY_CR_CONTROL</a> is not zero.</p>
8 [23:16]	DCR	R/cW	0x0	<p><b>Device Characteristics Register</b></p> <p>This field holds the BCR for the Host Controller while operating in Standby Controller mode.</p> <p>This field is used to automatically respond to the <a href="#">GETDCR</a> CCC, and to the <a href="#">ENTDAA</a> procedure if supported.</p> <p>This field is Read-Only while the <a href="#">STBY_CR_ENABLE_INIT</a> field in register <a href="#">STBY_CR_CONTROL</a> is not zero.</p>
15 [15:1]	PID_HI	R/cW	0x0	<p><b>Device Provisional ID High</b></p> <p>This field holds Bits[47:33] of the Controller's I3C PID, if required.</p> <p>This field is Read-Only while the <a href="#">STBY_CR_ENABLE_INIT</a> field in register <a href="#">STBY_CR_CONTROL</a> is not zero.</p> <p><i>Note: The Secondary Controller Logic shall return 1'b0 for PID Bit[32].</i></p>
1 [0]	RESERVED	-	-	-

#### 7.7.11.6 Standby Controller Device PID Low (STBY\_CR\_DEVICE\_PID\_LO) (+ 0x1C)

This register holds Bits[31:0] of the Host Controller's I3C PID while operating in Standby Controller mode, if the I3C PID is required (either by the **GETPID** CCC or by the **ENTDAA** procedure).

**Table 124 Standby Controller Device Characteristics (STBY\_CR\_DEVICE\_PID\_LO) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [31:0]	<b>PID_LO</b>	R/cW	0x0	<b>Device Provisional ID Low</b> This field holds Bits[31:0] of the Controller's I3C PID, if required. This field is Read-Only while the <b>STBY_CR_ENABLE_INIT</b> field in register <b>STBY_CR_CONTROL</b> is not zero.

### 7.7.11.7 Standby Controller Interrupt Status (STBY\_CR\_INTR\_STATUS) (+ 0x20)

This register reflects the status of outstanding interrupts relating to Standby Controller mode and the related I3C Secondary Controller Logic. This includes:

- Outstanding interrupts relating to Standby Controller mode operations, and
- Outstanding interrupts relating to transitions between Active Controller mode and Standby Controller mode (and vice versa).

The status fields are indicated as either:

- **R/W1C** (i.e., Write 1 to Clear) if cleared by software. R/W1C is used for original interrupt sources.

Or

- **R** (i.e., read only) if cleared by hardware based on other operations to the interrupt register sets.

The bit fields in this register act as Status Bits, according to the interrupt register model described in [Section 6.14](#).

These bits shall be set (i.e., the “Status bit set” action will occur) if the associated interrupt status or event is detected. The interrupt events or status in this register shall only be enabled for reporting (i.e., for assertion via trigger to register [STBY\\_CR\\_INTR\\_SIGNAL\\_EN](#)) under any of the following conditions:

- If the Host Controller is operating in Active Controller mode and is preparing to transition into Standby Controller mode (see [Section 6.17.2](#));
- If the Host Controller is operating in Standby Controller mode as a Secondary Controller role on the I3C Bus (see [Section 6.17.3](#)).

If an I3C Target Transaction Interface to Software is supported and enabled, then certain interrupt conditions shall also be reported for Write-Type transfers that are sent by the Active Controller of the I3C Bus but which the I3C Secondary Controller Logic did not autonomously service (see [Section 6.17.3](#));

- If the Host Controller is preparing to transition into Active Controller mode, and is ready to accept the Controller Role from the Active Controller (see [Section 6.17.4](#)); or
- If the Host Controller has just passed the Controller Role to another Controller-capable Device (i.e., a Secondary Controller) but the new Active Controller did not assert its Controller Role (i.e., an Error Type CE3 condition occurred), so the Host Controller was forced to reclaim the Controller Role and resume operation in Active Controller mode.

If the Host Controller is operating in Active Controller mode and is not preparing to pass the Controller Role to any indicated Controller-capable Device (i.e., to a Secondary Controller on the I3C Bus), then the interrupt events or status in this register shall be not be enabled for reporting in this register.

6917

**Table 125 Standby Controller Interrupt Status (STBY\_CR\_INTR\_STATUS) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
<b>13</b> [31:20]	RESERVED	—	—	—
<b>1</b> [19]	<b>CCC_FATAL_RSTDAA_ERR_STAT</b>	R/W1C	0x0	<b>CCC Fatal RSTDAA Error Status</b> The Host Controller received the Broadcast <b>RSTDAA</b> CCC and was not able to reset the Dynamic Address or receive a new Dynamic Address due to its implementation or configuration (see <b>Section 6.17.3.1</b> ).
<b>1</b> [18]	<b>CCC_UNHANDLED_NACK_STAT</b>	R/W1C	0x0	<b>CCC Unhandled NACK Status</b> The Host Controller was forced to NACK a Direct CCC which it was not able to handle because it did not support that CCC (see <b>Section 6.17.3.1</b> ).
<b>1</b> [17]	<b>CCC_PARAM_MODIFIED_STAT</b>	R/W1C	0x0	<b>CCC Parameter Modified Status</b> The Host Controller shall write 1'b1 to this field to indicate that it automatically handled a SET CCC successfully, for any of the Target parameters except Dynamic Address (see <b>Section 6.17.3.1</b> ).
<b>1</b> [16]	<b>STBY_CR_OP_RSTACT_STAT</b>	R/W1C	0x0	<b>Secondary Controller Operation Reset Action</b> The Host Controller shall write 1'b1 to this field to indicate that the Secondary Controller received a <b>RSTACT</b> CCC from the Active Controller, followed by the Target Reset Pattern (see <b>Section 6.17.3.1</b> ).
<b>1</b> [15]	RESERVED	—	—	—
<b>1</b> [14]	<b>STBY_CR_ACCEPT_ERR_STAT</b>	R/W1C	0x0	<b>Secondary Controller Transition Error Status</b> The Host Controller shall write 1'b1 to this field to indicate that Secondary Controller transition could not be completed due to an error (see <b>Section 6.17.4, Table 10</b> ).

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [13]	STBY_CR_ACCEPT_OK_STAT	R/W1C	0x0	<b>Secondary Controller Transition OK Status</b> The Host Controller shall write 1'b1 to this field to indicate that the Secondary Controller transition has completed successfully (see <a href="#">Section 6.17.4, Table 10</a> ).
1 [12]	STBY_CR_ACCEPT_NACKED_STAT	R/W1C	0x0	<b>Secondary Controller Transition NACKed</b> The Host Controller shall write 1'b1 to this field to indicate that it NACKed (i.e., did not accept) the <a href="#">GETACCCR</a> CCC directed to its Dynamic Address (see <a href="#">Section 6.17.4, Table 10</a> ).
1 [11]	STBY_CR_DYN_ADDR_STAT	R/W1C	0x0	<b>Secondary Controller Dynamic Address Status</b> The Host Controller shall write 1'b1 to this field to indicate any changes to the Dynamic Address, or to its "valid" status (in register <a href="#">STBY_CR_DEVICE_ADDRESS</a> , <a href="#">Section 7.7.11.2</a> ) as a result of actions taken by the Active Controller (see <a href="#">Section 6.17.1.3</a> ).
1 [10]	CRR_RESPONSE_STAT	R/W1C	0x0	<b>Controller Role Request Response Status</b> The Host Controller shall write 1'b1 to this field to indicate that the Secondary Controller Logic received a response to a Simple Controller Role Request from the Active Controller (see <a href="#">Section 6.17.3.3</a> ).
5 [9:4]	RESERVED	—	—	—
1 [3]	ACR_HANDOFF_ERR_M3_STAT	R/W1C	0x0	<b>Controller Role Handoff Error Type CE3 Recovery</b> The Secondary Controller Logic detected an Error Type CE3 condition, and the Host Controller returned to Active Controller Mode (see <a href="#">Section 6.17.2.2, Table 9</a> ).

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [2]	ACR_HANDOFF_ERR_FAIL_STAT	R/W1C	0x0	<p><b>Controller Role Handoff Error Due To Failure</b></p> <p>The Controller Role Handoff Procedure failed due to an error (i.e., cancellation) detected by the I3C Bus Controller Logic (see <b>Section 6.17.2.2, Table 9</b>).</p>
1 [1]	ACR_HANDOFF_OK_PRIMED_STAT	R/W1C	0x0	<p><b>Controller Role Handoff OK and Primed to Accept</b></p> <p>The Controller Role Handoff Procedure was successful and the Secondary Controller Logic is primed to automatically accept the Controller Role (see <b>Section 6.17.2.2, Table 9</b>).</p>
1 [0]	ACR_HANDOFF_OK_REMAIN_STAT	R/W1C	0x0	<p><b>Controller Role Handoff OK and Will Remain Secondary Controller</b></p> <p>The Controller Role Handoff Procedure was successful and the Secondary Controller Logic will remain active (i.e., is not primed to accept the Controller Role) (see <b>Section 6.17.2.2, Table 9</b>).</p>

#### 7.7.11.8 Standby Controller Interrupt Signal Enable (STBY\_CR\_INTR\_SIGNAL\_ENABLE) (+ 0x28)

6918 This register enables or disables signaling of outstanding interrupts received by the Host Controller.  
 6919 The bit fields in this register act as Interrupt Signal Enable Mask bits, according to the interrupt register  
 6920 model described in [Section 6.14](#). Writes to these fields shall enable or disable the delivery of an interrupt  
 6921 condition to the Host (i.e., the Interrupt trigger).

6922 **Table 126 Standby Controller Interrupt Signal Enable (STBY\_CR\_INTR\_SIGNAL\_ENABLE)  
 6923 Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
12 [31:20]	RESERVED	—	—	—
1 [19]	CCC_FATAL_RSTDAA_ERR_SIGNAL_EN	R/W	0x1	<b>Standby Controller Interrupt Signal Enable (various)</b> When set to 1'b1, and the corresponding interrupt status field is set in register <b>STBY_CR_INTR_STATUS</b> (see <a href="#">Section 7.7.11.7</a> ), the Host Controller shall assert an interrupt to the Host.
1 [18]	CCC_UNHANDLED_NACK_SIGNAL_EN	R/W	0x1	
1 [17]	CCC_PARAM_MODIFIED_SIGNAL_EN	R/W	0x0	
1 [16]	STBY_CR_OP_RSTACT_SIGNAL_EN	R/W	0x0	
1 [15]	RESERVED	—	—	
1 [14]	STBY_CR_ACCEPT_ERR_SIGNAL_EN	R/W	0x0	
1 [13]	STBY_CR_ACCEPT_OK_SIGNAL_EN	R/W	0x0	
1 [12]	STBY_CR_ACCEPT_NACKED_SIGNAL_EN	R/W	0x0	
1 [11]	STBY_CR_DYN_ADDR_SIGNAL_EN	R/W	0x0	
1 [10]	CRR_RESPONSE_SIGNAL_EN	R/W	0x0	
6 [9:4]	RESERVED	—	—	—
1 [3]	ACR_HANDOFF_ERR_M3_SIGNAL_EN	R/W	0x0	<b>Standby Controller Interrupt Signal Enable (various)</b> When set to 1'b1, and the corresponding interrupt status field is set in register <b>STBY_CR_INTR_STATUS</b> (see <a href="#">Section 7.7.11.7</a> ), the Host Controller shall assert an interrupt to the Host.
1 [2]	ACR_HANDOFF_ERR_FAIL_SIGNAL_EN	R/W	0x0	
1 [1]	ACR_HANDOFF_OK_PRIMED_SIGNAL_EN	R/W	0x0	

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [0]	ACR_HANDOFF_OK_REMAIN_SIGNAL_EN	R/W	0x0	

### 7.7.11.9 Standby Controller Interrupt Force (STBY\_CR\_INTR\_FORCE) (+ 0x2C)

This register is used to force a specific interrupt. It can be used for debugging purposes.

Writes to fields in this register act as Interrupt-force programming inputs, according to the interrupt register model described in [Section 6.14](#).

**Table 127 Interrupt Force (STBY\_CR\_INTR\_FORCE) Register**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
12 [31:20]	RESERVED	—	—	—
1 [19]	CCC_FATAL_RSTDAA_ERR_FORCE	W	0x0	<b>Standby Controller Interrupt Force (various)</b> For software testing, when set to 1'b1, forces the corresponding interrupt to be sent to the Host, if the corresponding fields are set in register <b>STBY_CR_INTR_SIGNAL_ENABLE</b> (see <a href="#">Section 7.7.11.8</a> ).
1 [18]	CCC_UNHANDLED_NACK_FORCE	W	0x0	
1 [17]	CCC_PARAM_MODIFIED_FORCE	W	0x0	
1 [16]	STBY_CR_OP_RSTACT_FORCE	W	0x0	
1 [15]	RESERVED	—	—	
1 [14]	STBY_CR_ACCEPT_ERR_FORCE	W	0x0	
1 [13]	STBY_CR_ACCEPT_OK_FORCE	W	0x0	
1 [12]	STBY_CR_ACCEPT_NACKED_FORCE	W	0x0	
1 [11]	STBY_CR_DYN_ADDR_FORCE	W	0x0	
1 [10]	CRR_RESPONSE_FORCE	W	0x0	
10 [9:0]	RESERVED	—	—	—

#### 7.7.11.10 Standby Controller CCC Auto-Response Config Get Capabilities (STBY\_CR\_CCC\_CONFIG\_GETCAPS) (+ 0x30)

This register contains data needed for the Secondary Controller Logic to auto-respond to the **GETCAPS** (Get Optional Feature Capabilities) CCC (Mandatory), for Format 2 with the **CRCAPS** Defining Byte.

For the read-only bits in these fields, the Secondary Controller Logic shall return an appropriate value: either based on the Host Controller's capabilities, or derived from configuration written via other registers.

The **R/cW** register fields are read-only from the Host while the Host Controller is operating in Standby Controller mode (i.e., while field **STBY\_CR\_ENABLE\_INIT** in register **STBY\_CR\_CONTROL** is set to a non-zero value; see *Section 7.7.11.1*). R/cW fields are write-only from the Host while the Secondary Controller Logic is disabled.

**Table 128 CCC Auto-Response Config Get Capabilities  
(STBY\_CR\_CCC\_CONFIG\_GETCAPS)**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
16 [31:16]	RESERVED	R	0x0	–
4 [15:12]	RESERVED	R	0x0	Reserved for future definition by MIPI Alliance ( <b>CRCAP2</b> , Bits[7:4])
4 [11:8]	F2_CRCAP2_DEV_INTERACT	R/cW or R	0x0 or IMPL	<b>GETCAPS CCC CRCAPS Byte 2 (CRCAP2, Bits[3:0])</b> Indicates I3C Device interaction capabilities when acting as an Active Controller. See I3C Specification [ <b>MIPI02</b> ] at <i>Section 5.1.9.3.19, Table 38</i> .
5 [7:3]	RESERVED	R	0x0	Reserved for future definition by MIPI Alliance ( <b>CRCAP1</b> , Bits[5:3])
3 [2:0]	F2_CRCAP1_BUS_CONFIG	R/cW or R	0x0 or IMPL	<b>GETCAPS CCC CRCAPS Byte 1 (CRCAP1, Bits[2:0])</b> Indicates I3C Bus configuration capabilities when acting as an Active Controller. See I3C Specification [ <b>MIPI02</b> ] at <i>Section 5.1.9.3.19, Table 37</i> .

#### 7.7.11.11 Standby Controller CCC Auto-Response Config Target Reset Action (STBY\_CR\_CCC\_CONFIG\_RSTACT\_PARAMS) (+ 0x34)

This register contains the timing parameters that the Secondary Controller Logic will use to autonomously service the response to the Active Controller sending the **RSTACT** Broadcast or Direct GET CCC (per *Section 6.17.3.1*) with certain Target Reset-related Defining Byte values (see I3C Specification [**MIPI02**] at *Section 5.1.9.3.26*).

**Table 129 CCC Auto-Response Config Target Reset Action  
(STBY\_CR\_CCC\_CONFIG\_RSTACT\_PARAMS)**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [31]	RESET_DYNAMIC_ADDR	R/cW or R	1'b1	<p><b>Reset Dynamic Address after Target Reset</b></p> <p>If set to 1'b1, then the Secondary Controller Logic must clear its Dynamic Address in register <b>STBY_CR_DEVICE_ADDR</b> (see <i>Section 7.7.11.2</i>) after receiving a Target Reset Pattern that followed a Broadcast or Direct SET <b>RSTACT</b> CCC sent to the Dynamic Address, with Defining Byte 0x01 or 0x02.</p> <p>Requires support for Dynamic Address Assignment with at least one supported method, such as the <b>ENTDAA</b> CCC, with field <b>DAA_ENTDAA_ENABLE</b> set to 1'b1 in register <b>STBY_CR_CONTROL</b> (<i>Section 7.7.11.1</i>).</p> <p>If field <b>ACR_FSM_OP_SELECT</b> in register <b>STBY_CR_CONTROL</b> is set to 1'b1, then this field shall be cleared (i.e., readiness to accept the Controller Role shall be revoked) with this Target Reset Pattern.</p>
7 [30:24]	RESERVED	R/W	0x0	–
8 [23:16]	RESET_TIME_TARGET	R/cW	0x0	<p><b>Time to Reset Target</b></p> <p>For Direct GET CCC, this field is returned for Defining Byte 0x82.</p>
8 [15:8]	RESET_TIME_PERIPHERAL	R/cW	0x0	<p><b>Time to Reset Peripheral</b></p> <p>For Direct GET CCC, this field is returned for Defining Byte 0x81.</p>
8 [7:0]	RST_ACTION	R	0x0	<p><b>Defining Byte of the RSTACT CCC</b></p> <p>Contains the Defining Byte received with the last Direct SET CCC sent by the Active Controller.</p>

### 7.7.12 Target-Specific (ID = 0xB0...0xBF)

6944 Capability-specific read-only registers with Extended Capability IDs 0xB0 through 0xBF are reserved for  
6945 definition in future versions of this Specification.

### 7.7.13 Vendor-Specific Extended Capabilities (ID = 0xC0...0xCF)

6946 Details of the Capability-specific read-only registers with Extended Capability IDs 0xC0 through 0xCF  
6947 (Vendor-Specific) are vendor-specific and not detailed in this Specification.

## 8 Data Structures

This Section describes several data structures used by the Host Controller.

**Note:**

*Some of these data structures are implemented as HCI Registers, and as a result do not appear in the memory that the software Driver allocates.*

### 8.1 Device Address Table (DAT)

The DAT table stores the Device Addresses of Target Devices attached to the I3C Bus, plus several control and information fields per Device. The DAT may be either implemented as part of the HCI Register Set (i.e., may be implemented via regular registers) or placed in Driver-allocated memory as part of Device Context (see [Section 6.3.3](#)). In this section the DAT is presented in structure form for easier understanding.

When used for Transfer Commands and automatic response to I3C Bus conditions, the DAT must contain one entry for each attached Target Device with an assigned Dynamic Address. A DAT entry must also be allocated for each assigned Group Address, if supported (see [Section 6.4.3](#)).

Each entry in the DAT has the structure shown in [Table 130](#). For Command types that require DAT table entries, DAT table entries are indexed by the Command's **DEV\_INDEX** field. Register **DAT\_SECTION\_OFFSET** (see [Section 7.4.11](#)) in the Capabilities registers section (see [Section 7.3.1](#)) indicates the offset to the DAT and the number of entries in the DAT.

The DAT is populated by software, and hardware uses it during the Dynamic Address Assignment procedure. Per-Device prioritization of IBIs from Target Devices can be set via Driver software control over the Dynamic Address Arbitration procedure.

**Note:**

*If the DAT is placed in Driver-allocated memory, then software should use the Internal Control Command with sub-command 0x3 for **Device Context Update** (see [Section 8.4.2.3](#)) to inform the Host Controller of any changes to DAT entries, as and when such changes are written by software.*

Each valid DAT entry contains the Device's Static Address (if applicable) and its Dynamic Address (if configured). This specification supports 7-bit Target Device Addresses. If the Device is addressed via Dynamic Address, then the Static Address shall not be used. The Static Address shall only be used for I3C Target Devices that support Dynamic Address Assignment with the **SETDASA** CCC, and for I<sup>2</sup>C Target Devices (which must always have Static Addresses).

The size of each DAT entry is 2 DWORDs.

6976

**Table 130 Device Address Table Structure**

<b>Size [Bits]</b>	<b>Field Name</b>	<b>Memory Access</b>	<b>Reset Value</b>	<b>Description</b>
<b>5</b> [63:59]	RESERVED	—	—	—
<b>8</b> [58:51]	AUTOCMD_HDR_CODE	R/W	0x0	<p><b>Device Auto-Command HDR Command Code</b>          Specifies Auto-Command Read Command Code. Only valid if Read is executed in HDR Mode.</p> <p><b>For HDR-DDR Mode and HDR-Ternary Mode:</b>          If the value is below 0x80, then the Host Controller shall use 0x80 for the HDR Command Code.</p>
<b>3</b> [50:48]	AUTOCMD_MODE	R/W	0x0	<p><b>Device Auto-Command Mode</b>          Indicates the Mode and speed for the Auto-Command Read.          Only valid for the Device in I3C Mode (see the DEVICE field in the DAT Table entry indexed by field DEV_INDEX).</p> <p><b>Values for I3C Mode:</b></p> <ul style="list-style-type: none"> <li>• <b>0x0: I3C SDR0</b>            Standard SDR Speed, <math>f_{SCL}</math> Max (up to 12.5 MHz)</li> <li>• <b>0x1–0x4: I3C SDR1–SDR4</b>            Reduced data rates (see <b>Section 7.1.1.1</b> of the I3C TCRI Specification [<a href="#">MIPI06</a>])</li> <li>• <b>0x5: I3C HDR-TSx</b>            HDR-Ternary Mode. Selection of HDR-TSP vs. HDR-TSL depends on the value of field I2C_DEV_PRESENT in register <b>HC_CONTROL</b>.</li> <li>• <b>0x6: I3C HDR-DDR</b>            HDR Double Data Rate Mode</li> <li>• <b>0x7: Reserved</b></li> </ul>
<b>8</b> [47:40]	AUTOCMD_VALUE	R/W	0x0	<p><b>Device Auto-Command IBI Mandatory Byte Value</b>          Value of the IBI Mandatory Byte that will trigger an Auto-Command Read transaction on the I3C Bus</p>
<b>8</b> [39:32]	AUTOCMD_MASK	R/W	0x0	<p><b>Device Auto-Command Mask</b>          Mask for the IBI Mandatory Byte that will trigger an Auto-Command Read transaction on the I3C Bus. See Note below.</p>
<b>1</b> [31]	DEVICE	R/W	0x0	<p><b>Device Type</b>  <b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>0x0: I3C</b>: I3C Device</li> <li>• <b>0x1: I2C</b>: I<sup>2</sup>C Device</li> </ul>

Size [Bits]	Field Name	Memory Access	Reset Value	Description
2 [30:29]	DEV_NACK_RETRY_CNT	R/W	0x0	<p><b>Device NACK Retry Count</b> Device-specific retry count</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>0x0:</b> No retries on NACK (does not apply to Direct CCCs; see <a href="#">Section 6.3</a> of the I3C TCRI Specification [<a href="#">MIPI06</a>]).</li> <li>• <b>0x1–0x3:</b> Up to 1–3 Retries on NACK</li> </ul>
3 [28:26]	RING_ID	R/W	0x0	<p><b>Device Ring Group ID</b> Used to send an IBI from a specific Device to the appropriate Ring Bundle.</p>
2 [25:24]	RESERVED	R/W	0x0	—
8 [23:16]	DYNAMIC_ADDRESS	R/W	0x0	<p><b>Device I3C Dynamic Address</b></p> <ul style="list-style-type: none"> <li>• <b>Bits[22:16]:</b> Shall contain the Dynamic Address. A Dynamic Address of 7'h00 indicates that this DAT entry is disabled.</li> <li>• <b>Bit[23]:</b> Is the Parity Bit, per the I3C Specification [<a href="#">MIPI02</a>] at <a href="#">Section 5.1.4.2</a>, computed and updated by the software Driver (see <a href="#">Section 8.1.2</a>).</li> </ul>
1 [15]	TS	R/W	0x0	<p><b>Device IBI Timestamp</b> Enables or disables IBI timestamping for a specific Device.</p> <p><b>Note:</b> <i>The IBI Status Descriptor for each IBI event indicates whether or not the individual IBI event was actually timestamped.</i></p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>1'b0: NO_TS:</b> The Controller shall not timestamp IBIs from this Device with Controller Timestamps</li> <li>• <b>1'b1: TS:</b> The Controller shall timestamp IBIs for this Device with Controller Timestamps</li> </ul>
1 [14]	CRR_REJECT	R/W	0x0	<p><b>Device In-Band Controller Role Request Reject</b> Controls whether this Host Controller, when operating in the Active Controller role, shall accept vs. reject Controller Role Requests from the Secondary Controller Device indicated in this DAT entry.</p> <p>This bit is only valid if the Host Controller declares Standby Controller Capability.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>1'b0: ACCEPT:</b> ACK Controller Role Requests from the Secondary Controller</li> <li>• <b>1'b1: REJECT:</b> NACK any Controller Role Request, and send the auto-disable CCC (i.e., <a href="#">DISEC</a>)</li> </ul>

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [13]	IBI_REJECT	R/W	0x0	<p><b>Device In-Band Interrupt Request Reject</b>  Controls whether the Host Controller, when operating in the Active Controller role, shall accept vs. reject In-Band Interrupt Requests from the Target Device indicated by this DAT entry.  The Host Controller shall ignore this field if the Target IBI credit counting mechanism is both supported and currently enabled (per <a href="#">Section 6.9.5</a>).  <b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: ACCEPT: ACK In-Band Interrupt Requests from this Target</li> <li>• 1'b1: REJECT: NACK any IBI, and send the auto-disable CCC (i.e., <a href="#">DISEC</a>)</li> </ul>
1 [12]	IBI_PAYLOAD	R/W	0x0	<p><b>Device IBI Payload</b>  Indicates whether IBIs from this Device have a Data Payload. This field reflects the IBI Payload bit in the Device's Bus Characteristics Register (BCR).  During IBI handling for this Device, the Controller shall use this field to determine whether or not to drive reception of the IBI Data Payload. Data continuation is indicated by the T-Bit.  <b>Values:</b></p> <ul style="list-style-type: none"> <li>• 1'b0: NO_PAYLOAD: IBIs from this Device do not carry a Data Payload</li> <li>• 1'b1: PAYLOAD: IBIs from this Device do carry a Data Payload</li> </ul>
5 [11:7]	RESERVED	R/W	0x0	–
7 [6:0]	STATIC_ADDRESS	R/W	0x0	<p><b>Device Static Address</b>  I3C / I<sup>2</sup>C Static Address</p>

6977

**Note:**6978  
6979

If both fields **AUTOCMD\_VALUE** and **AUTOCMD\_MASK** are set to a value of 0x0 for an entry, then that entry shall match an IBI with any Mandatory Data Byte, per [Section 6.11](#).

### 8.1.1 DAT Usage for Transfer Commands

The DAT entry's scope of usage requires software to assign (i.e., to allocate) DAT entries for all known Target Devices that will be used for Transfer Commands.

- Every Target Device shall have its own DAT entry for Transfer Commands.

Software shall include an index into the DAT table for all Transfer Commands that address a specific Target Device by its Dynamic Address, or for Write-Type Transfer Commands that address a Group Address (if configured and supported).

Broadcast CCCs do not use DAT entries, per **Section 6.7**.

Additionally, most commands used for Dynamic Address Assignment also require DAT entries for all target Devices that are found, or are expected to be found. Specifically, the Address Assignment Commands (see **Section 8.4.1**) require DAT entries for the **ENTDAA** and **SETDASA** CCCs.

**Note:**

*This puts an upper limit of 32 possible unique I3C Addresses, including the sum of assigned Dynamic Addresses and Group Addresses, on an I3C Bus that is used with an I3C Controller that complies with this I3C HCI Specification. If additional I3C Addresses are assigned, then such I3C Devices cannot all be addressed simultaneously by such an I3C Controller. Note that this upper limit might be reduced for Host Controller implementations that implement a smaller DAT (e.g., a reduced number of DAT entries as a Register Set).*

### 8.1.2 DAT Addressing and Parity Bit

Software shall provide the Dynamic Address and Parity Bit, written into field **DYNAMIC\_ADDRESS** of a particular DAT entry before using it. This field must be written before any use of an Address Assignment Command, or any Transfer Command. The Dynamic Address shall be written into Bits[22:16], and the Parity Bit shall be written into Bit[23].

Any DAT entry with a Dynamic Address of 7'h00 shall be considered to be disabled by the Host Controller, and shall not be valid for Transfer Commands. Such a DAT entry shall also not be matched for any incoming IBIs, as this Dynamic Address is a reserved I3C Address, per the I3C Specification [**MIPI02**] at **Section 5.1.2.2.5**.

The Parity Bit is used for Dynamic Address Assignment with the **ENTDAA** CCC, per the I3C Specification [**MIPI02**] at **Section 5.1.4.2**. The Parity Bit shall be calculated using an inverse XOR of the Dynamic Address:

$$\text{PARITY} = \sim\text{XOR}(\text{ADDR}[6:0])$$

Software shall calculate the Parity Bit, and the Host Controller may choose to either use the Parity Bit written into Bit[23], or else ignore Bit[23] and calculate the Parity Bit directly from the Dynamic Address (i.e., from Bits[22:16]).

**Note:**

*The implementer may determine whether the Host Controller will use the provided value for the Parity Bit in Bit[23], or automatically calculate the correct Parity Bit value based on the Dynamic Address alone.*

## 8.2 Device Characteristic Table (DCT)

The DCT table captures the Device Characteristics (PID, BCR, and DCR) and assigned Dynamic Address of each Device on the I3C Bus that participates in the Dynamic Address Allocation (ENTDAA) procedure.

The DCT table is considered to be either part of the HCI Register Set (i.e., implemented as regular registers), or else placed in Driver-allocated memory as part of Device Context (see [Section 6.3.3](#)). In this section it is presented in structure form for easier understanding.

**Note:**

*If the DCT is placed in Driver-allocated memory, then the Driver should not write to the DCT entries.*

Register [DCT\\_SECTION\\_OFFSET](#) (see [Section 7.4.12](#)), located in the Capabilities registers section (see [Section 7.3.1](#)), indicates the offset to the DCT and the number of entries in the DCT.

The DCT contains one entry for each Device, and each entry has the structure shown in [Table 131](#).

**Table 131 Device Characteristic Table Structure**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
<b>24</b> [127:104]	RESERVED	—	—	—
<b>8</b> [103:96]	DYNAMIC_ADDRESS	R	0x0	<b>Device I3C Dynamic Address</b> Includes Parity Bit.
<b>16</b> [95:80]	RESERVED	—	—	—
<b>8</b> [79:72]	BCR	R	0x0	<b>Device BCR</b> Value of Device's I3C Bus Characteristics Register.
<b>8</b> [71:64]	DCR	R	0x0	<b>Device DCR</b> Value of Device's I3C Device Characteristics Register.
<b>16</b> [63:48]	RESERVED	—	—	—
<b>16</b> [47:32]	PID_LO	R	0x0	<b>Device Provisional ID Low</b> Bits [15:0] of Device's I3C PID.
<b>32</b> [31:0]	PID_HI	R	0x0	<b>Device Provisional ID High</b> Bits [48:16] of Device's I3C PID.

## 8.3 Transfer Descriptor

7027 **Note:**

7028     *This section applies only for Host Controller implementations that implement DMA Mode, and thus*  
7029     *connect to a System Bus that supports memory-mapped IO and DMA.*

7030     The Transfer Descriptor is a structure put on Command/Response Rings in order to perform the transfer.

7031     The Transfer Descriptor is composed of a single Command Descriptor structure, and either a Data Buffer  
7032     Pointer or an array of Data Buffer Pointers:

- 7033     • If the buffer for a requested transfer is organized as a single block of contiguous physical memory, then  
7034         the Data Buffer Pointer within the Transfer Descriptor points directly to that memory block. Its field **BLP**  
7035         shall have a value of 1'b0.
- 7036     • For Scatter-Gather Buffers, which are composed of multiple, usually smaller, non-contiguous blocks of  
7037         physical memory, the array of Data Buffer Pointers is used. A list of multiple Data Buffer Pointers is  
7038         provided, with each Data Buffer Pointer pointing to a separate memory blocks. The Data Buffer Pointer  
7039         within the Transfer Descriptor shall act as a Data Buffer List Pointer that points to the array of Memory  
7040         Descriptor structures (see **Section 8.7**), and its field **BLP** shall have a value of 1'b1.

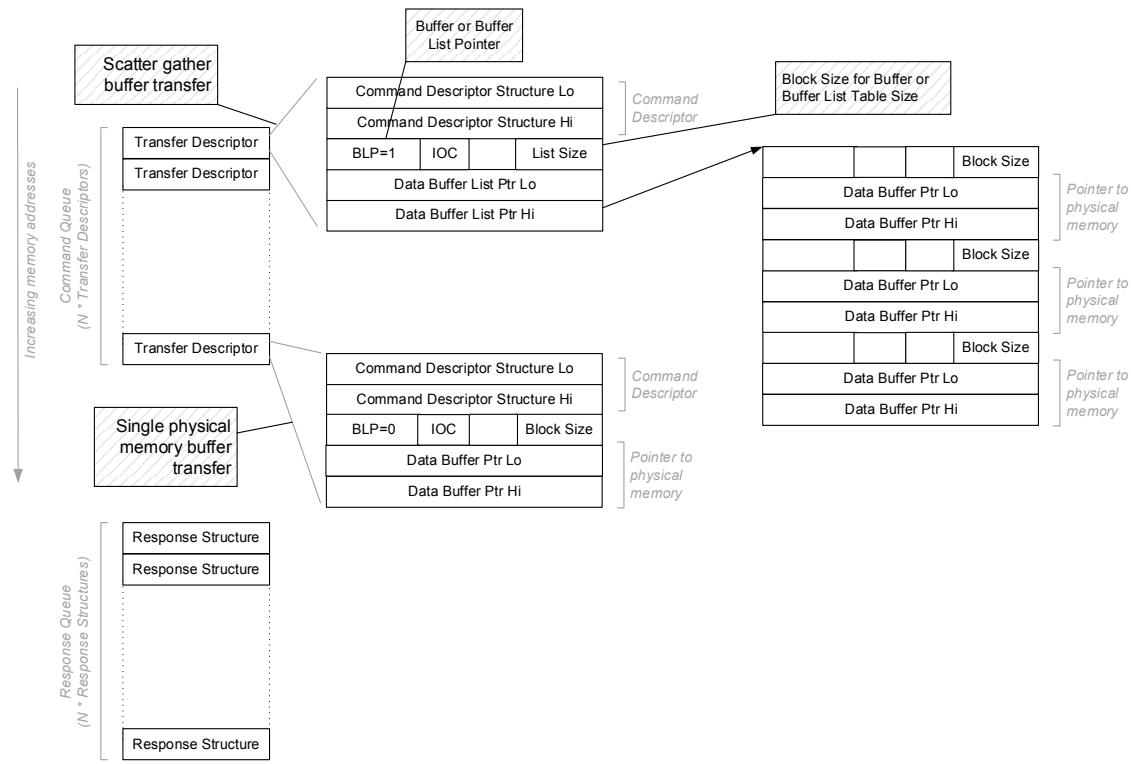
7041     The Transfer Descriptor is a minimum of 5 DWORDs in size: 2 DWORDs for the Command Descriptor per  
7042     **Section 6.7**, and 3 DWORDs for the Data Buffer Descriptor (including its Data Buffer Pointer), per  
7043     **Section 8.3.1**.

7044 **Note:**

7045     *The first DWORD (i.e., DW0) of the Transfer Descriptor is always the low DWORD of the Command*  
7046     *Descriptor. Per Section 7.6.10.1, an implementer may define a larger Transfer Descriptor size,*  
7047     *indicated in field XFER\_STRUCT\_SIZE of register CR\_SETUP. If so, then the first 2 DWORDs shall*  
7048     *consist of the Command Descriptor, the next 3 DWORDs shall consist of the Data Buffer*  
7049     *Descriptor, and any additional DWORDs may be used for vendor-specific extensions.*

7050     Figure 49 shows one example of Scatter-Gather Buffers (i.e., using an array of Data Buffer Pointers), and  
7051     one example of contiguous memory transfers (i.e., using just a single Data Buffer Pointer).

7052

**Figure 49 Use of Transfer Descriptor**

### 8.3.1 Data Buffer Descriptor

The Data Buffer Descriptor is a structure that is directly used in a Transfer Descriptor. This structure contains a pointer in fields **BUFFER\_PTR\_HI** / **BUFFER\_PTR\_LO**. Field **BLP** indicates whether this pointer is:

- A direct pointer to a single Data Buffer for the entire transfer (i.e., a single region of physically contiguous memory); or
- A pointer to an array of Data Buffer Pointers that are set up as a Scatter-Gather List (i.e., an array of Memory Descriptor structures), each having their own with a pointer to a block of physical memory to be used as a portion of the Data Buffer for the transfer.

**Note:**

*Scatter-Gather Lists always use the Memory Descriptor structure (see [Section 8.7](#)). The Data Buffer Descriptor is always contained within the Transfer Descriptor.*

7065

**Table 132 Data Buffer Descriptor Structure**

<b>Size [Bits]</b>	<b>Field Name</b>	<b>Memory Access</b>	<b>Reset Value</b>	<b>Description</b>
<b>32 [95:64]</b>	<b>BUFFER_PTR_HI</b>	R/W	0x0	<b>Data Buffer Pointer High</b>
<b>32 [63:32]</b>	<b>BUFFER_PTR_LO</b>	R/W	0x0	<b>Data Buffer Pointer Low</b>
<b>1 [31]</b>	<b>BLP</b>	R/W	0x0	<p><b>Buffer Vs. List Pointer</b>            Indicates whether the Data Buffer Pointer (i.e., fields <b>BUFFER_PTR_HI</b> / <b>BUFFER_PTR_LO</b>) points directly to the Data Buffer, or to a List of buffers describing the Data Buffer.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>1'b0: BUFFER:</b> Pointer points directly to physical memory used as the Data Buffer</li> <li>• <b>1'b1: LIST:</b> Pointer points to physical memory comprising array of Memory Descriptors (i.e., a Scatter-Gather List that might not point to physically contiguous memory)</li> </ul>
<b>1 [30]</b>	<b>IOC</b>	R/W	0x0	<p><b>Transfer Interrupt on Completion</b>            Indicates whether the Host Controller shall assert the <b>TRANSFER_COMPLETION_STAT</b> interrupt (see <b>Section 7.6.10.4</b>) after it completes this transfer (see <b>Section 6.6.2.2</b> and <b>Section 6.8.2</b>).</p>
<b>14 [29:16]</b>	<b>RESERVED</b>	R/W	0x0	–
<b>16 [15:0]</b>	<b>BLOCK_SIZE or LIST_SIZE</b>	R/W	0x0	<p><b>Data Buffer Block Size or List Size</b></p> <ul style="list-style-type: none"> <li>• <b>If field BLP contains 1'b0:</b> Buffer Size (in Bytes)</li> <li>• <b>If field BLP contains 1'b1:</b> Scatter-Gather List Size (in Entries)</li> </ul> <p>When a Scatter-Gather List is used, all but the last list entry's memory block must be sized to a multiple of DWORD (i.e., the last entry's memory block can be a non-multiple). The Host Controller shall pad, to align to a DWORD boundary.</p> <p><b>Note:</b>  <i>If field BLP=1'b0, then this field should be set to a non-zero value, except for valid transfers with no data bytes that only require an ACK of a Dynamic Address (such as any Direct CCCs that do not have payload defined).</i></p>

## 8.4 Command Descriptor

The write-only Command Descriptor structure defines a transaction, including its parameters, and is sent by the Driver to schedule a command to a Device(s) on the I3C Bus while the Host Controller is operating in Active Controller mode.

The method for writing a Command Descriptor depends on the operating mode (see [Section 6.8](#)):

- **PIO Mode:** A Command Descriptor is added to the Command Queue via writes to the Command Queue Port (per [Section 6.8.1](#))
- **DMA Mode:** A Command Descriptor is included in the Transfer Descriptor, and the Transfer Descriptor is placed on the Command Ring (per [Section 6.8.2](#))

The Command Descriptor is 64 bits (2 DWORDs) in length, and supports a number of common transfer types (see [Section 6.7](#)).

All Command Descriptors can be grouped into the supported Command Types shown in [Table 133](#). This specification defines a Command Descriptor structure for each listed Command Type, at the indicated Section. [Table 133](#) also shows the value of the **CMD\_ATTR** field, for each listed Command Type.

- The **Address Assignment** Command type is used for assigning Dynamic Addresses to I3C Targets.
- The **Internal Control** Command type is used for controlling the Host Controller itself, not for I3C Transfer Commands. However, specific sub-commands may affect the way that the I3C Bus Controller processes certain Transfer Commands.

**Table 133 Supported Command Types for Command Descriptor**

Code	Command Type	CMD_ATTR	Defined in Section	Note
A	Address Assignment Command	0x2	<a href="#">8.4.1</a>	–
M	Internal Control Command	0x7	<a href="#">8.4.2</a>	–
I	Immediate Data Transfer Command	0x1	I3C TCRI Specification, <a href="#">Section 7.1.2.1</a>	1
R	Regular Transfer Command	0x0	I3C TCRI Specification, <a href="#">Section 7.1.2.2</a>	1
C	Combo Transfer Command	0x3	I3C TCRI Specification, <a href="#">Section 7.1.2.3</a>	1

**Note:**

1. See the *I3C TCRI Specification [MIPI06]* at [Section 7.1.1](#) and [Section 7.1.2](#) and their sub-sections for the normative definitions of these Transfer Commands, and at [Section 8.4.3](#) for additional details specific to Host Controller implementations.

**Figure 50** and **Figure 51** provide a high-level overview of the Command Types supported by the Command Descriptor for all supported I3C Commands, showing one row per Command Type. For many Command Types, field **DEV\_INDEX** holds an index to a specific DAT entry for all transfer operations.

- The formats for Transfer Command Types are largely compatible with the formats defined in version 1.0 of this Specification. Field **DEV\_INDEX** is defined for all transfer Commands that use DAT entries for the Dynamic Address of the indicated Target Device. If Group Addresses are supported, then a DAT entry shall also be used for each such Group Address.

**Note:**

*In version 1.1 of this I3C HCI Specification, field **WROC** was previously named **ROC**. This is a name change only; there is no technical change to the meaning of this field.*

Fields for Immediate Data Transfer Command																																
Bits in DWORD																																
DWORD 1 (N+32)																																
DWORD 0 (N+0)																																
TOC   WROC   RnW   DATA_BYTE_4   DATA_BYTE_3   DATA_BYTE_2   DATA_BYTE_1 or DEF_BYTE																																
MODE   DTT   RSVD   DEV_INDEX   ?   CMD   TID   CMD_ATTR																																
Fields for Regular Transfer Command																																
Bits in DWORD																																
DWORD 1 (N+32)																																
DWORD 0 (N+0)																																
TOC   WROC   RnW   DATA_LENGTH   RESERVED   DEF_BYTE																																
MODE   DBP   SRE   RSVD   DEV_INDEX   ?   CMD   TID   CMD_ATTR																																
Fields for Combo Transfer Command																																
Bits in DWORD																																
DWORD 1 (N+32)																																
DWORD 0 (N+0)																																
TOC   WROC   RnW   DATA_LENGTH   RESERVED   OFFSET or SUBOFFSET																																
MODE   DBP   SRE   16/8   FPM   DLP   R   DEV_INDEX   ?   CMD   TID   CMD_ATTR																																

**Figure 50 Overview of Supported Command Types for Command Descriptor (Part 1)**

Fields for Address Assignment Command																																	
Bits in DWORD																																	
DWORD 1 (N+32)																																	
DWORD 0 (N+0)																																	
RESERVED																																	
TOC   WROC   DEV_COUNT   RSVD   DEV_INDEX   R   CMD   TID   CMD_ATTR																																	
Fields for Internal Control Command																																	
Bits in DWORD																																	
DWORD 1 (N+32)																																	
DWORD 0 (N+0)																																	
VENDOR_SPECIFIC																																	
MIPI_RESERVED (for sub-commands)																																	
MIPI_CMD   VIP   TID   CMD_ATTR																																	

**Figure 51 Overview of Supported Command Types for Command Descriptor (Part 2)**

#### 8.4.1 Address Assignment Command

This section defines the Command Descriptor structure for I3C Address Assignment commands. This Command Type supports the modal flow for I3C Dynamic Address Assignment, using the **ENTDAA** CCC (see *Section 5.1.4.2* and *Section 5.1.9.3.4* of the I3C Specification [*MIPI02J*]); and also supports the command to set an I3C Target's Dynamic Address from its assigned Static Address, using the **SETDASA** CCC (see *Section 5.1.4.2* and *Section 5.1.9.3.10* of the I3C Specification [*MIPI02J*]).

The Address Assignment Command relies on the Driver to provide a Dynamic Address in one or more DAT entries as part of Device Context (see *Section 6.3.3* and *Section 8.1*). It is the only method for assigning a Dynamic Address using the **SETDASA** and **ENTDAA** CCCs (per *Section 6.4*).

**Note:**

*The Address Assignment Command does not support the **SETAASA** Broadcast CCC.*

**Table 134 Address Assignment Command Structure**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [63:32]	RESERVED	—	—	—
1 [31]	TOC	W	0x0	<b>Terminate on Completion</b> Controls what Bus condition to issue after the Address Assignment Command completes. <b>Values:</b> <ul style="list-style-type: none"><li>• 1'b0: RESTART: Issue Repeated START (Sr) at end of transfer</li><li>• 1'b1: STOP: Issue Stop (P) at end of transfer</li></ul> <b>Note:</b> <i>This field must be set to 1'b1 for ENTDAA. It is meaningful for SETDASA transfers.</i>
1 [30]	WROC	W	0x0	<b>Response on Completion</b> Controls whether Response Status is sent after successful completion of the Address Assignment. The successful completion shall be read from register <b>RESPONSE_QUEUE_PORT</b> . Upon unsuccessful assignment the Response Status shall always be sent. <b>Note:</b> <i>For this field, a value of 1'b0 is only valid for PIO Mode. In DMA Mode, the Host Controller shall always generate a Response Descriptor, and software shall always set this field to 1'b1.</i> <b>Values:</b> <ul style="list-style-type: none"><li>• 1'b0: NOT_REQUIRED: Response Status is not required</li><li>• 1'b1: REQUIRED: Response Status is required</li></ul>
4 [29:26]	DEV_COUNT	W	0x0	<b>Device Count</b> Indicates the number of Devices that a Dynamic Address shall be assigned to. The DAT entries are processed starting at index <b>DEV_INDEX</b> for up to <b>DEV_COUNT</b> times. The processed DAT entries shall be contiguous in memory.

<b>Size [Bits]</b>	<b>Field Name</b>	<b>Memory Access</b>	<b>Reset Value</b>	<b>Description</b>
<b>5</b> [25:21]	RESERVED	—	—	—
<b>5</b> [20:16]	DEV_INDEX	W	0x0	<p><b>Device Index</b>            Indicates the DAT table index for the Target Device being assigned an address via Dynamic Address Assignment.            Software shall write a Dynamic Address into each DAT entry that will be used with this command type, as well as a Static Address (when used with the <a href="#">SETDASA</a> CCC).</p>
<b>1</b> [15]	RESERVED	—	—	—
<b>8</b> [14:7]	CMD	W	0x0	<p><b>Transfer Command CCC Value</b>            Specifies CCC code indicating whether Address Assignment uses ENTDAA or SETDASA commands.            The field comprises the entire command code (<a href="#">ENTDAA</a> or <a href="#">SETDASA</a>)</p>
<b>4</b> [6:3]	TID	W	0x0	<p><b>Transaction ID</b>            Used as a tag for this command.</p>
<b>3</b> [2:0]	CMD_ATTR	W	0x0	<p><b>Command Attributes</b>            Command Type, defining the format of the other fields.  <b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>0x2: ADDR_ASSGN_CMD:</b> Address Assignment Command</li> <li>• All other values are either defined for other Command Types, or reserved for future use.</li> </ul>

#### 8.4.1.1 Usage for DAA with ENTDAAC CCC

The Address Assignment Command supports Dynamic Address Assignment with the **ENTDAAC** CCC and its procedure, for I3C Target Devices that support Dynamic Address Assignment using the **ENTDAAC** procedure (per the I3C Specification [*MIPI02*] at *Section 5.1.4.2*).

This variant of the Address Assignment Command requires a standard DAT and DCT having at least one entry in each table (although more entries are strongly recommended). If the Host Controller supports the DAT and DCT as part of Device Context that must be allocated in Host System memory, then the appropriate base address registers must first be programmed by the Driver (see *Section 7.4.19* and *Section 7.4.20*) as part of initializing the Host Controller for operation (per *Section 6.1*).

Before initiating Dynamic Address Assignment with the **ENTDAAC** CCC and its procedure, the Driver shall allocate DAT entries for the number of Target Devices that are expected to be found. These might be a set of contiguous entries in the DAT; if so, then the Driver can prepare these in a sequence and use a single Address Assignment Command to assign them all in a single pass. Alternatively, this might be a single entry in the DAT, used either for a single expected command, or for multiple invocations of the same command (subject to some restrictions).

For each DAT entry, the Driver shall set the **DEVICE** field to indicate the Device's type (i.e., I3C), and then set the **DYNAMIC\_ADDRESS** field to the desired Dynamic Address. The Driver shall not set an assigned Static Address in the **STATIC\_ADDRESS** field, as no Static Address is assigned to such Target Devices. If DMA Mode is used, then the Driver may also choose a specific Ring Bundle ID to use for any incoming IBI Requests by writing a Ring Bundle ID that has been configured with a Ring IBI Status/Data Pair (per *Section 6.6.3*) into field **RING\_ID**.

The Driver should also initialize the DCT index for each invocation of an Address Assignment Command with ENTDAAC, by writing a value of 0x0 to field **TABLE\_INDEX** in register **DCT\_SECTION\_OFFSET** (*Section 7.4.12*).

- For most cases the default index value of 0x0 should be written, as this allows the Host Controller to start at the first DCT entry for each invocation of an Address Assignment Command with ENTDAAC. A value of 0x0 is strongly recommended, and the Driver should generally not override this recommendation.
- Use of other index values might be possible for certain use cases. However, the Host Controller shall increment the value of field **TABLE\_INDEX** upon each successful assignment of a Dynamic Address to a Target Device during the ENTDAAC procedure. In some cases, the index value might wrap around back to the initial value (i.e., index 0x0) after it increments past the last DCT entry (i.e., if the number of found Devices is greater than **TABLE\_SIZE** minus **TABLE\_INDEX**).

To use the **ENTDAA** CCC and its procedure, the Driver shall prepare a Command Descriptor of the appropriate format, with the following specified field values:

- For Bits[31:0]:

- Field **CMD\_ATTR** shall contain the value 0x2 (**ADDR\_ASSGN\_CMD**).
- Field **TID** may have any value chosen by software.
- Field **CMD** shall contain 0x07, the assigned Common Command Code value for **ENTDAA** (per the I3C Specification [**MIPI02**] at **Section 5.1.9.3**).
- Field **DEV\_INDEX** shall indicate the DAT table index, for the first (or only) DAT entry that contains a Dynamic Address configured earlier:
  - If only a single DAT entry is expected to be used, then the Driver shall provide the index to the DAT entry, and shall then write a value of 0x1 into field **DEV\_COUNT**.
  - If several contiguous DAT entries are expected to be used, then the Driver shall provide the index to the first DAT entry, and shall then write a value into field **DEV\_COUNT** for the number of contiguous DAT entries (i.e., the number of Target Devices to configure with **ENTDAA**).
- Field **WROC** shall contain the value 1'b1 (i.e., Response Status is required).
- Field **TOC** shall contain the value 1'b1 (i.e., issue the STOP [P] condition at the end of the transfer).

- For Bits[63:32]:

- These bits are reserved, and shall contain all-zero values.

Once the Host Controller receives this Command Descriptor, it shall validate that it matches the specified format. If a Command Descriptor is received that does not match the specified format, then it shall be rejected and field **ERR\_STATUS** of the Response Descriptor shall contain the value 0xA (**NOT\_SUPPORTED**), per **Section 8.5**. If the Command Descriptor does match the specified format, then the Host Controller's Controller Logic shall start the Dynamic Address Assignment modal flow using the **ENTDAA** CCC with Broadcast CCC Framing in SDR Mode. The Controller logic shall assign the Dynamic Addresses to any I3C Target Devices that respond to the Dynamic Address Assignment procedure, in the same order as the provided DAT entries. The Controller Logic shall start at the DAT entry indicated by **DEV\_INDEX** and proceed incrementally, until it encounters a failure (i.e., a NACK) or until the **ENTDAA** procedure has completed.

The **ENTDAA** procedure shall end either when no additional I3C Target Devices that have not been assigned a Dynamic Address respond to the modal flow, or when the Controller Logic has assigned all Dynamic Addresses from the range of contiguous DAT entries, as indicated by field **DEV\_COUNT** in the Command Descriptor.

**Note:**

*Dynamic Address Assignment with the **ENTDAA** CCC is not guaranteed to be predictable, since the Arbitration phase of the **ENTDAA** procedure affects the order in which I3C Target Devices might win the arbitration. A changing I3C Bus configuration could mean that the same inputs to DAT entries might not cause the Host Controller's I3C Bus Controller Logic to assign the same Dynamic Addresses to the same Target Devices, if new Target Devices are added to an I3C Bus that might affect the Arbitration during the **ENTDAA** procedure.*

If the Address Assignment Command was accepted and processed without any error, then field **ERR\_STATUS** of the Response Descriptor shall contain a value of 0x0 (**SUCCESS**) for either of the following two cases:

1. During the ENTDAA procedure, the I3C Bus Controller Logic was able to assign the Dynamic Addresses from some or all of the indicated DAT entries, for a number of found Target Devices less than or equal to the value of field **DEV\_COUNT**, with no additional eligible Target Devices remaining without a Dynamic Address.
  - In this case, field **DATA\_LENGTH** of the Response Descriptor shall contain a value of 0x0 to indicate no remaining Target Devices (i.e., that the **ENTDAA** procedure was not terminated early).
2. During the ENTDAA procedure, the I3C Bus Controller Logic was able to assign the Dynamic Addresses from all of the indicated DAT entries, for a number of found Target Devices equal to the value of field **DEV\_COUNT**, and was forced to terminate the **ENTDAA** procedure since there were additional eligible Target Devices remaining without a Dynamic Address.
  - In this case, field **DATA\_LENGTH** of the Response Descriptor shall contain the value 0x1, to indicate at least one remaining Target Device. Note that the true count of remaining Target Devices cannot be detected by the I3C Bus Controller Logic, since the process of Arbitration during the **ENTDAA** procedure indicates only that some additional Target Devices are present, not how many there are.
  - In this case, the Driver should prepare additional DAT entries (i.e., with unused Dynamic Addresses) and then send a new Address Assignment Command to start a new **ENTDAA** procedure, in order to assign Dynamic Addresses to any remaining Target Devices.

**Note:**

*The case where the I3C Bus Controller Logic might terminate the **ENTDAA** procedure early might happen frequently, for situations where the Host Controller has a smaller DAT and must make decisions about which Target Devices might need to be assigned a dedicated DAT entry. This case might also happen frequently for situations where the Driver might not know the total count of Target Devices on the I3C Bus, and might also need to examine each found Target Device and assign it a Dynamic Address based on its PID, BCR, and DCR values, that would account for its relative priority on the I3C Bus. However, the Driver would first need to assign a temporary Dynamic Address using the DAT entry, in order to use the Host Controller and the **ENTDAA** procedure to learn about the Target Device. For this situation, the Driver would use repeated invocations of the Address Assignment Command for the **ENTDAA** procedure, with field **DEV\_COUNT** set to a value of 0x1; followed by other Transfer Commands with the **SETNEWDA** CCC, to re-assign the temporary Dynamic Address to a longer-term Dynamic Address.*

In both cases, the Driver should read the current value from field **TABLE\_INDEX** in register **DCT\_SECTION\_OFFSET** ([Section 7.4.12](#)), and compare with the original value written to the same field, in order to determine the number of successful assignments. The Driver should also read the Dynamic Address, PID, BCR, and DCR values from the appropriate DCT entries (or from the single DCT entry) that were written by the Host Controller for each successful assignment, in order to save the values in memory.

**Note:**

*If the original and current values from field **TABLE\_INDEX** are identical, then the ENTDAA procedure did not assign any Dynamic Addresses. The Driver should detect this result and handle it appropriately.*

If a Target Device did not ACK the Dynamic Address that the I3C Bus Controller Logic assigned to it, then the Host Controller shall stop the ENTDAA procedure and indicate this as an error. In such a situation, field **ERR\_STATUS** of the Response Descriptor shall contain a value of 0x5 (**NACK**). Note that a partial success might still generate a NACK error, with some DCT entries being written with Dynamic Address, PID, BCR, and DCR values for those assignments that succeeded before the I3C Bus Controller Logic received a NACK.

#### 8.4.1.2 Usage for DAA with SETDASA CCC

The Address Assignment Command supports Dynamic Address Assignment with the **SETDASA** CCC, for I3C Target Devices that have an assigned Static Address and also support the **SETDASA** CCC.

This variant of the Address Assignment Command requires a standard DAT having at least one entry in the table, although more DAT entries are usually recommended for most use cases. If the Host Controller supports the DAT as part of Device Context that must be allocated in Host System memory, then the appropriate base address registers must first be set by the Driver (see *Section 7.4.19* and *Section 7.4.20*) as part of initializing the Host Controller for operation (per *Section 6.1*).

Before initiating Dynamic Address Assignment with the **SETDASA** CCC, the Driver shall allocate DAT entries for such Target Devices that are known to have assigned Static Addresses. These might be a set of contiguous entries in the DAT; if so, the Driver can prepare these in a sequence, and use a single Address Assignment Command to assign them all in a single pass.

For each DAT entry, the Driver shall set the **DEVICE** field to indicate the Device's type (i.e., I3C), and then set the **DYNAMIC\_ADDRESS** field to the desired Dynamic Address. The Driver shall then set the assigned Static Address in the **STATIC\_ADDRESS** field. If DMA Mode is used, then the Driver may also choose a specific Ring Bundle ID to use for any incoming IBI Requests, by writing a Ring Bundle ID that has been configured with a Ring IBI Status/Data Pair (as specified in *Section 6.6.3*) into field **RING\_ID**.

To use the **SETDASA** CCC, the Driver shall prepare a Command Descriptor of the appropriate format, with the following specified field values:

- For Bits[31:0]:

- Field **CMD\_ATTR** shall contain the value 0x2 (**ADDR\_ASSGN\_CMD**).
- Field **TID** may have any value chosen by software.
- Field **CMD** shall contain the value the assigned Common Command Code value for **SETDASA** (i.e., 0x87; see the I3C Specification [*MIPI02*] at *Section 5.1.9.3*).
- Field **DEV\_INDEX** shall indicate the DAT table index, for the first (or only) DAT entry that contains the Dynamic Address and Static Address configured earlier:
  - If this is for a single Target Device with a single DAT entry, then the Driver shall provide the index to the DAT entry, and shall then write a value of 0x1 into field **DEV\_COUNT**.
  - If this is for several Target Devices with a range of contiguous DAT entries, then the Driver shall provide the index to the first DAT entry, and then write a value into field **DEV\_COUNT** for the number of contiguous DAT entries (i.e., the number of Target Devices to configure with **SETDASA**).
- Field **WROC** shall contain the value 1'b1 (i.e., Response Status is required).
- Field **TOC** shall contain the value 1'b1 (i.e., issue the STOP [P] condition at the end of the transfer).

- For Bits[63:32]:

- These bits are reserved, and shall contain all-zero values.

Once the Host Controller receives this Command Descriptor, it shall validate that it matches the specified format. If a Command Descriptor is received that does not match the specified format, then it shall be rejected and field **ERR\_STATUS** of the Response Descriptor shall contain the value 0xA (**NOT\_SUPPORTED**), per *Section 8.5*. If the Command Descriptor does match the specified format, then the Host Controller's I3C Bus Controller Logic shall start the **SETDASA** CCC with Direct CCC Framing in SDR Mode, using the values of Static Addresses and Dynamic Addresses from the provided DAT entries. The I3C Bus Controller Logic shall start at the DAT entry indicated by **DEV\_INDEX** and proceed incrementally, until it encounters a failure (i.e., a NACK) or until all DAT entries have been processed and all Dynamic Addresses successfully assigned.

If the Address Assignment Command was accepted and processed without any error, then field **ERR\_STATUS** of the Response Descriptor shall contain a value of 0x0 (**SUCCESS**). However, if any Target Devices did not ACK the **SETDASA** CCC, then field **ERR\_STATUS** of the Response Descriptor shall contain a value of 0x5 (**NACK**). The Host Controller shall also indicate how many Target Devices were remaining from the Address Assignment Command that were not assigned a Dynamic Address.

- If field **DATA\_LENGTH** has a value of 0x0, then all Target Devices indicated by field **DEV\_COUNT** were assigned the Dynamic Address in the DAT entry (or entries).
- If field **DATA\_LENGTH** has a non-zero value, then one or more Target Devices did not receive their Dynamic Address. Field **DATA\_LENGTH** therefore gives an indirect indication of a Target Device that did not respond to the **SETDASA** CCC and did not ACK a Static Address:
  - The indicated DAT entry at index **DEV\_INDEX + DEV\_COUNT - DATA\_LENGTH** shall be the DAT entry for the Target Device that returned a NACK for the **SETDASA** CCC.
  - If field **DATA\_LENGTH** is greater than 1, then the subsequent DAT entries after this entry were not processed by the Host Controller, since the I3C Bus Controller Logic ended the Direct CCC framing after receiving the NACK. Target Devices indicated by these subsequent DAT entries have not received a Dynamic Address, and must still be addressed by their assigned Static Addresses.

#### 8.4.2 Internal Control Command

This section defines the Command Descriptor structure for controlling the Host Controller itself (not I3C Transfer Commands).

The same Internal Control Command Structure (see *Table 135*) is used for several sub-commands which are detailed in the sub-sections below. The desired sub-command is selected with the value in field **MIPI\_CMD**:

- **0x1: Ring Bundle Lock/Unlock**, see *Section 8.4.2.1*
- **0x2: Broadcast Address Enable/Disable**, see *Section 8.4.2.2*
- **0x3: Device Context Update**, see *Section 8.4.2.3*
- **0x4: Target Reset Pattern**, see *Section 8.4.2.4*
- **0x5: Controller SDA Recovery or Bus Reset Procedure**, see *Section 8.4.2.5*
- **0x6: Enable End Transfer Termination and HDR Mode Configuration**, see *Section 8.4.2.6*
- **0x7: Controller Role Handoff Procedure with GETACCCR CCC** (new for I3C HCI v1.2), see *Section 8.4.2.7*
- **0xD: Attempt Dead Bus Recovery** (new for I3C HCI v1.2), see *Section 8.4.2.8*

Sub-command specific sub-fields appear in field **MIPI\_RESERVED** which is at Bits[31:12]. In other words, the format and usage of field **MIPI\_RESERVED** will depend upon the value of field **MIPI\_CMD**.) The sub-fields for each sub-command are detailed in *Table 136* through *Table 140* and *Table 144*.

7302

**Table 135 Internal Control Command Structure**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [63:32]	VENDOR SPECIFIC	W	0x0	<b>Vendor Specific</b> The definition of this field is vendor-specific.
20 [31:12]	MIPI_RESERVED	W	0x0	<b>Command Dependent Area (MIPI Alliance Reserved)</b> Format and usage depends upon value of field MIPI_CMD; see <b>Table 136</b> through <b>Table 140</b> .
4 [11:8]	MIPI_CMD	W	0x0	<b>MIPI Alliance Command</b> Indicates the MIPI Alliance sub-command for the Internal Command Descriptor (see <b>Section 8.4.2.1</b> through <b>Section 8.4.2.5</b> ). <b>Values:</b> <ul style="list-style-type: none"> <li>• 0x0: NoOp (i.e., performs no transfer)</li> <li>• 0x1: Ring Bundle Lock/Unlock</li> <li>• 0x2: Broadcast Address Enable/Disable</li> <li>• 0x3: Device Context Update</li> <li>• 0x4: Target Reset Pattern</li> <li>• 0x5: Controller SDA Recovery or Bus Reset Procedure</li> <li>• 0x6: Enable Early Termination and HDR Mode Configuration</li> <li>• 0x7: Controller Role Handoff Procedure</li> <li>• 0xD: Attempt Dead Bus Recovery</li> <li>• All other values: Reserved for future use</li> </ul> <b>Note:</b> <i>Every MIPI Alliance sub-command supported by the Host Controller (excluding value 0x0) shall be indicated as supported in bitmask field MIPI_CMDS_SUPPORTED in register INT_CTRL_CMDS_EN (see <b>Section 7.4.16</b>).</i>
1 [7]	VENDOR_INFO_PRESENT	W	0x0	<b>Vendor Info Present</b> Indicates whether a Vendor-Specific extension is present.
4 [6:3]	TID	W	0x0	<b>Transaction ID</b> Identification tag for the command.
3 [2:0]	CMD_ATTR	W	0x0	<b>Command Attribute</b> Command Type, defining the format of the other fields. <b>Values:</b> <ul style="list-style-type: none"> <li>• 0x7: INTERNAL_CONTROL: Internal Control command.</li> <li>• All other values are defined for other Command Types, or reserved for future use.</li> </ul>

#### 8.4.2.1 Sub-Command 0x1: Ring Bundle Lock/Unlock

If field **MIPI\_CMD** in the Internal Control Command's Command Descriptor contains 0x1, then the sub-command is Ring Bundle Lock/Unlock. This sub-command is used by software to lock the Ring Controller's arbitration logic to only process Transfer Descriptors for a specific Ring Bundle's Command/Response Ring Pair, and not process any enqueued Transfer Descriptors for any other valid Ring Bundle.

This sub-command is valid for DMA Mode only, and shall only be used if the Host Controller has multiple Ring Bundles. While locked, the Ring Controller shall switch to other valid Ring Bundles until software subsequently unlocks the arbitration logic. See *Section 6.6.5.2*.

This sub-command uses Bit[12] of field **MIPI\_RESERVED** for sub-field **ON\_OFF**.

**Table 136 Internal Control Command Fields for Ring Bundle Lock/Unlock**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
<b>19</b> [31:13]	<b>MIPI_RESERVED</b>	W	0x0	MIPI Alliance Reserved area
<b>1</b> [12]	<b>ON_OFF</b>	W	0x0	<b>Ring Bundle Lock / Unlock</b> Allows the Host Controller command processing to be locked to a specific Ring Bundle's Command and Response Rings for this I3C Bus instance. <b>Values:</b> <ul style="list-style-type: none"> <li>• 1'b0: Lock Off</li> <li>• 1'b1: Lock On</li> </ul>
<b>4</b> [11:8]	<b>MIPI_CMD</b>	W	0x0	<b>MIPI Alliance Command</b> <b>Value: 0x1</b> for this sub-command
<b>8</b> [7:0]	<b>Common to all Internal Control Command Sub-command Types</b> (See <b>Table 135</b> )			

#### 8.4.2.2 Sub-Command 0x2: Broadcast Address Enable/Disable

If field **MIPI\_CMD** in the Internal Control Command's Command Descriptor contains 0x2, then the sub-command is Broadcast Address Enable/Disable. This sub-command is used by software to inform the Host Controller to either enable or disable automatic transmission of the I3C Broadcast Address and its header (i.e., START, 7'h7E, W followed by Repeated START) for every Transfer Command that indicates a Private Write or Private Read in SDR Mode, and would otherwise drive a START condition (i.e., starting from Bus Free condition or longer).

This sub-command uses Bit[12] of field **MIPI\_RESERVED** for sub-field **ON\_OFF**.

This sub-command acts similarly to field **IBA\_INCLUDE** in register **HC\_CONTROL** (see *Section 7.4.2*), but acts only on a particular operating context.

The particular operating context depends on the operating mode:

- **For PIO Mode:** The operating context is the PIO Command Queue. Software may send this sub-command as an Internal Control Command, by enqueueing the Command Descriptor into the Command Queue Port (per *Section 6.8.1*). The Host Controller shall keep the persistent setting (i.e., Broadcast Address transmission enabled or disabled) for subsequent Transfer Commands.

- **For DMA Mode:** The operating context is the Command Ring for the Ring Bundle that receives the sub-command. Software may send this sub-command as an Internal Control Command, by enqueueing a Transfer Descriptor containing this Command Descriptor into the Command Ring (per *Section 6.8.2*). The Host Controller shall keep the persistent setting (i.e., Broadcast Address transmission enabled or disabled) for subsequent Transfer Commands on the same Ring Bundle.

If the Host Controller supports multiple Ring Bundles (per *Section 6.6.5*) then it shall keep persistent settings for each Command Ring in a Ring Bundle, and apply them on switching back to that Ring Bundle to process subsequent Transfer Descriptors (per *Section 6.6.5.1*).

**Note:**

*In many cases, using this sub-command might be preferable to setting or clearing field **IBA\_INCLUDE**, as the update of the Broadcast Address transmission setting using this sub-command shall be “in-line” with the Transfer Commands that follow the update.*

**Table 137 Internal Control Command Structure: Fields for 0x7E Sub Command**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
19 [31:13]	<b>MIPI_RESERVED</b>	W	0x0	<ul style="list-style-type: none"><li>• MIPI Alliance Reserved area</li></ul>
1 [12]	<b>ON_OFF</b>	W	0x0	<b>Broadcast Address Enable / Disable</b> Enables or disables automatic transmission of the I3C Broadcast Header after every Start condition on this I3C Bus instance. <b>Values:</b> <ul style="list-style-type: none"><li>• 1'b0: Broadcast Address Disable</li><li>• 1'b1: Broadcast Address Enable</li></ul>
4 [11:8]	<b>MIPI_CMD</b>	W	0x0	<b>MIPI Alliance Command</b> <b>Value:</b> 0x2 for this sub-command
8 [7:0]	<b>Common to all Internal Control Command Sub-command Types</b> (See <b>Table 135</b> )			

#### 8.4.2.3 Sub-Command 0x3: Device Context Update

If field **MIPI\_CMD** in the Internal Control Command's Command Descriptor contains 0x3, then the sub-command is Device Context Update. This sub-command is used by software to inform the Host Controller that the Host has updated DAT entries in Host system memory, since the Host Controller is not able to automatically determine this (i.e., as would be possible with the Host writing to the DAT implemented in the Host Controller's register map; see [Section 8.1](#)).

This sub-command is valid only for Host Controllers that support DMA (i.e., a Memory Access Engine) that enables access to Host system memory for Device Context, and that reports a DAT Table Offset (i.e., field **TABLE\_OFFSET** in register **DAT\_SECTION\_OFFSET**; see [Section 7.4.11](#)) with a value of 12'h000, indicating that the Host Controller requires the Driver to provide Device Context in Host system memory, specifically for the DAT.

This sub-command's use of field **MIPI\_RESERVED** is:

- Bits[13:12] for sub-field **OP\_TYPE**
- Bits[18:14] for sub-field **INDEX**

**Table 138 Internal Control Command Fields for Device Context Update**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
13 [31:19]	<b>MIPI_RESERVED</b>	W	0x0	<ul style="list-style-type: none"> <li>• MIPI Alliance Reserved area</li> </ul>
5 [18:14]	<b>INDEX</b>	W	0x0	<b>DAT Index</b> Indicates the DAT table index for the Target Device being addressed with the command. Assumes DAT is stored in Host system memory (as Device Context).
2 [13:12]	<b>OP_TYPE</b>	W	0x0	<b>DAT Entry Operation Type</b> Indicates whether the indexed DAT entry in Host system memory is being added, removed, or changed in this command. <b>Values:</b> <ul style="list-style-type: none"> <li>• 0x0: DAT entry is being added</li> <li>• 0x1: DAT entry is being removed</li> <li>• 0x2: DAT entry is being changed</li> <li>• 0x3: Reserved for future use</li> </ul>
4 [11:8]	<b>MIPI_CMD</b>	W	0x0	<b>MIPI Alliance Command</b> <b>Value: 0x3</b> for this sub-command
8 [7:0]	<b>Common to all Internal Control Command Sub-command Types</b> (See <a href="#">Table 135</a> )			

**Note:**

The Host Controller might also choose to update any internal state associated with DAT entries in Host system memory on a periodic schedule or other basis, even if the Host does not explicitly use this sub-command. The method that a Host Controller uses to store and manage any internal state for such DAT entries is considered to be an implementation-specific detail, and is not defined in this I3C HCI Specification

#### 8.4.2.4 Sub-Command 0x4: Target Reset Pattern

If field **MIPI\_CMD** in the Internal Control Command Descriptor contains 0x4, then the sub-command is Target Reset Pattern. This sub-command is used by software to either drive a Target Reset Pattern, or to inform Host Controller of the start of a sequence of commands for several types of flows involving the **RSTACT** CCC and the Target Reset Pattern (see the I3C Specification [**MIPI02**] at **Section 5.1.11.2** and **Section 5.1.9.3.26**).

This command and any subsequent Transfer Commands for the **RSTACT** CCC (for certain flows that use the Target Reset Pattern after the **RSTACT** CCC) shall follow the steps defined in **Section 6.15.1**. For such flows that also involve the **RSTACT** CCC, this command shall instruct the Host Controller to enter a Critical Section, where the Host Controller shall handle this command and subsequent Transfer Commands according to special rules until the end of the sequence of commands, or unless the command sequence is ended prematurely (i.e., due to an error by software or a delay from the Host).

This sub-command uses Bits[13:12] of field **MIPI\_RESERVED** for sub-field **RESET\_OP\_TYPE**.

**Table 139 Internal Control Command for Target Reset Pattern**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
18 [31:14]	<b>MIPI_RESERVED</b>	W	0x0	MIPI Alliance Reserved area
2 [13:12]	<b>RESET_OP_TYPE</b>	W	0x0	<b>Target Reset Operation Type</b> Indicates whether the Target Reset Pattern should be issued immediately, or following one or more Transfer Descriptors containing <b>RSTACT</b> CCCs to configure Target Reset action(s) for one or more Target Devices (see <b>Section 6.15.1</b> ). Refer to the I3C Specification [ <b>MIPI02</b> ] at <b>Section 5.1.11</b> for details on Target Reset Pattern and its use with <b>RSTACT</b> CCC. <b>Values:</b> <ul style="list-style-type: none"><li><b>0x0:</b> Drive single Target Reset Pattern (i.e., Reset I3C Peripheral action, or Wakeup action).</li><li><b>0x1:</b> Reserved for future use.</li><li><b>0x2:</b> Prepare to send one or more Command Descriptors with <b>RSTACT</b> CCC; enter the Critical Section for special command handling; conditionally drive a single Target Reset Pattern on success. Full operational details are defined in <b>Section 6.15.1.2</b>.</li><li><b>0x3:</b> Leave the Critical Section initiated by operation type 0x2 above.</li></ul>
4 [11:8]	<b>MIPI_CMD</b>	W	0x0	<b>MIPI Alliance Command</b> <b>Value: 0x4</b> for this sub-command
8 [7:0]	<b>Common to all Internal Control Command Sub-command Types</b> (See <b>Table 135</b> )			

**Note:**

For Target Reset Pattern Command option 0x0 (see [Table 139](#)), refer to [Section 6.15.1.1](#) for details.

For Target Reset Pattern Command options 0x2 and 0x3 (see [Table 139](#)), refer to [Section 6.15.1.2](#) for recommended usage and full operational details.

#### **8.4.2.5 Sub-Command 0x5: Controller SDA Recovery or Bus Reset Procedure**

If field **MIPI\_CMD** in the Internal Control Command's Command Descriptor contains 0x5, then the sub-command is Controller SDA Recovery or Bus Reset Procedure. This sub-command is used by software to drive one of several recovery or reset procedures that are defined in the I3C Specification (see the I3C Specification [[MIPICO2](#)] at [Section 5.1.10.2](#)) and strongly recommended for various situations where a Target is unresponsive, SDA might be stuck, or an unexpected crash or reset has occurred.

This sub-command's use of field **MIPI\_RESERVED** is:

- Bits[15:12] for sub-field **REC\_RESET\_PROC**
- Bits[19:16] for sub-field **SCL\_SPEED**

**Table 140 Internal Control Command for Controller SDA Recovery or Bus Reset Procedure**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
<b>12</b> [31:20]	<b>MIPI_RESERVED</b>	W	0x0	MIPI Alliance Reserved area
<b>4</b> [19:16]	<b>SCL_SPEED</b>	W	0x0	<p><b>SCL Clock Speed</b>            The speed at which to drive SCL clock changes for the procedure indicated by field <b>REC_RESET_PROC</b> (i.e., Bits[15:12]).            For additional details, see <a href="#">Section 7.1.1.1</a> of the I3C TCRI Specification [<a href="#">MIPICO6</a>].</p> <p><b>Values for I3C Mode:</b></p> <ul style="list-style-type: none"> <li>• <b>0x0–0x4:</b> All I3C SDR supported data rates</li> <li>• <b>0x6:</b> I3C HDR-DDR speed supported</li> </ul> <p><b>Values for I<sup>2</sup>C Mode:</b></p> <ul style="list-style-type: none"> <li>• <b>0x0–0x4:</b> I<sup>2</sup>C at supported data rates</li> </ul> <p><b>All other values not listed above:</b> Reserved for future use. See <a href="#">Section 6.15.2</a> for more information.</p>

Size [Bits]	Field Name	Memory Access	Reset Value	Description
4 [15:12]	REC_RESET_PROC	W	0x0	<p><b>Recovery or Reset Procedure</b>  Selects the Recovery or Reset procedure.  Refer to the I3C Specification [<a href="#">MIPI02</a>] at <b>Section 5.1.10.2</b> for details on various recovery and reset procedures for an I3C Controller.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>0x0:</b> Recover SDA stuck Low from I<sup>2</sup>C transaction.</li> <li>• <b>0x1:</b> Recover SDA stuck Low or High from I3C SDR transaction.</li> <li>• <b>0x2:</b> Recover SDA stuck Low or High from I3C HDR-DDR transaction.</li> <li>• <b>0x4:</b> Force a STOP Condition; if SDA and SCL are both high, drives a START, then 7'h7E, then STOP.</li> <li>• <b>0x5:</b> Use CE2 error handling for a non-responsive I3C Target, by sending HDR Exit Pattern after NACK of Private read/write.</li> <li>• <b>0x7:</b> I3C Bus Reset with SDA High or Low, after Controller crash or unexpected Controller reset.</li> <li>• <b>All other values:</b> Reserved for future use.  See <b>Section 6.15.2</b> for more information.</li> </ul>
4 [11:8]	MIPI_CMD	W	0x0	<p><b>MIPI Alliance Command</b>  <b>Value: 0x5</b> for this sub-command</p>
8 [7:0]	<b>Common to all Internal Control Command Sub-command Types</b> (See <b>Table 135</b> )			

#### 8.4.2.6 Sub-Command 0x6: Enable End Transfer Termination and HDR Mode Configuration

7387 **Note:**

7388     *This sub-command is new for this version of the I3C HCI Specification, and is required if the Host*  
7389     *Controller supports either the End Transfer Termination capabilities for selected HDR Modes or*  
7390     *support for Monitoring Devices on the I3C Bus. However, if the Host Controller does not support*  
7391     *any of these capabilities, then this sub-command shall not be implemented.*

7392     If field **MIPI\_CMD** in the Internal Control Command's Command Descriptor contains 0x6, then the sub-  
7393     command is used to configure the I3C Bus Controller Logic to enable special handling of End Transfer  
7394     Termination for selected HDR Modes and for Monitoring Devices. This sub-command is used by software  
7395     as part of command sequences that use I3C transfers where these capabilities will be used by I3C Targets  
7396     on the Bus.

7397     Use of this sub-command for HDR Mode Configuration requires the Host to have previously configured  
7398     the relevant I3C Targets to support the indicated configuration, by sending the **ENDXFER** CCC with  
7399     appropriate options (see the I3C Specification [**MIPI02**] at **Section 5.1.9.3.25**). The options provided to this  
7400     sub-command must match the configuration of the Target(s) that will be addressed in subsequent I3C  
7401     transfers that follow this sub-command.

7402     For specific information about the configuration of HDR Mode transfers, refer to the I3C Specification  
7403     [**MIPI02**]:

- 7404     • For HDR-DDR Mode, see **Section 5.2.2.3.4**
- 7405     • For HDR-Ternary Modes, see **Section 5.2.3.3.2**

7406     Use of this sub-command for Monitoring Device Early Termination shall apply to various I3C Modes,  
7407     including SDR Mode, HDR-DDR Mode, and HDR-Ternary Modes (see the I3C Specification [**MIPI02**] at  
7408     **Section 5.1.12.2.1**). For this type of configuration, the I3C Bus Controller sends the **ENDXFER** CCC to  
7409     enable Monitoring Devices for subsequent transfers, and configures the I3C Bus Controller Logic to  
7410     appropriately handle situations where a Monitoring Device terminates the data transfer.

7411     The Host may send one or more sub-commands of this type with different configuration parameters as part  
7412     of a command sequence; additionally, such sub-commands must precede any transfers in the HDR Mode  
7413     that will rely on such parameters (i.e., before the I3C Bus enters that HDR Mode). After the I3C Bus  
7414     Controller Logic accepts each sub-command, the provided parameters will remain in force until the next  
7415     STOP condition on the I3C Bus.

7416 **Note:**

7417     *After the next STOP condition (which typically happens at the end of the command sequence), the*  
7418     *special handling is disabled: the I3C Bus Controller Logic shall return to its default configuration*  
7419     *parameters. However, implementers may choose to define one or more optional Extended*  
7420     *Capability structures with registers to set the default configuration for such HDR Modes. If so, then*  
7421     *such configuration parameters will then become the default after the next STOP condition (i.e.,*  
7422     *instead of disabling the special handling).*

7423     This sub-command's use of field **MIPI\_RESERVED** is:

- 7424     • Bits[15:12] for sub-field **END\_XFER\_TYPE**
- 7425     • Bits[23:16] for sub-field **END\_XFER\_CONFIG**

7426  
7427

**Table 141 Internal Control Command for Enable End Transfer Termination and HDR Mode Configuration**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
8 [31:24]	MIPI_RESERVED	W	0x0	MIPI Alliance Reserved area
8 [23:16]	END_XFER_CONFIG	W	0x0	<b>End Transfer Configuration Parameters</b> Contains the parameters used with the <a href="#">ENDXFER</a> CCC to either configure selected Targets for HDR Mode Configuration, or configure the Bus to support Monitoring Device Early Termination.
4 [15:12]	END_XFER_TYPE	W	0x0	<b>End Transfer Type</b> Selects the type of configuration that will apply to subsequent transfers. Refer to the I3C Specification <a href="#">[MIPI02]</a> at <a href="#">Section 5.1.9.3.25</a> for details on the <a href="#">ENDXFER</a> CCC. <b>Values:</b> <ul style="list-style-type: none"> <li>• <b>0x0:</b> Reserved for future use.</li> <li>• <b>0x1:</b> Apply the parameters in field <a href="#">END_XFER_CONFIG</a> to subsequent transfers in HDR-DDR Mode. Parameters in field <a href="#">END_XFER_CONFIG</a> must match previously configured Targets (i.e., after using the <a href="#">ENDXFER</a> CCC with Defining Bytes 0xF7 and 0xAA).</li> <li>• <b>0x2:</b> Apply the parameters in field <a href="#">END_XFER_CONFIG</a> to subsequent transfers in HDR-Ternary Modes. Parameters in field <a href="#">END_XFER_CONFIG</a> must match previously configured Targets (i.e., after using the <a href="#">ENDXFER</a> CCC with Defining Bytes 0x7F and 0x55).</li> <li>• <b>0x3:</b> Enable support for Monitoring Device Early Termination Capability, and apply the parameters in field <a href="#">END_XFER_CONFIG</a> to subsequent transfers. The I3C Bus Controller shall send the <a href="#">ENDXFER</a> CCC with Defining Byte 0xFC, using the parameters in field <a href="#">END_XFER_CONFIG</a>, before any subsequent transfers in this command sequence.</li> <li>• <b>All other values:</b> Reserved for future use. See <a href="#">Section 6.8.6</a> for more information.</li> </ul>
4 [11:8]	MIPI_CMD	W	0x0	<b>MIPI Alliance Command</b> <b>Value:</b> 0x6 for this sub-command
8 [7:0]	Common to all Internal Control Command Sub-command Types (See <a href="#">Table 135</a> )			

#### 8.4.2.7 Sub-Command 0x7: Controller Role Handoff Procedure with GETACCCR CCC

7428 **Note:**

7429     *This sub-command is new for this version of the I3C HCI Specification, and is required if the Host*  
7430     *Controller supports Standby Controller mode (i.e., with Secondary Controller Logic).*

7431 If field **MIPI\_CMD** in the Internal Control Command's Command Descriptor contains 0x7, then this sub-  
7432 command is Controller Role Handoff Procedure with the **GETACCCR** CCC. The Driver uses this sub-  
7433 command to pass the Controller Role to a chosen Controller-capable Device (i.e., a Secondary Controller  
7434 Device) on the I3C Bus. This sub-command shall only be used while in the special transitional state to  
7435 prepare for Controller Role Handoff which is used to prepare for entering Standby Controller mode (per  
7436 **Section 6.17.2**).

7437 The Host Controller shall only accept this sub-command in this transitional state, and shall reject this sub-  
7438 command if it is not operating in the transitional state. Additionally, the Host Controller shall always  
7439 generate a Response Descriptor, and shall return one of several error codes in field **ERR\_STATUS** depending  
7440 on the results of the sub-command and its attempt to drive the **GETACCCR** CCC to the chosen Secondary  
7441 Controller Device, per **Table 143**. If the Host Controller returns 0x0 (**SUCCESS**) in field **ERR\_STATUS** of the  
7442 Response Descriptor, then this typically indicates a successful transition from Active Controller mode to  
7443 Standby Controller mode; however, other interrupt conditions are typically sent along with the change of  
7444 roles on the I3C Bus.

7445 As the Host Controller processes this sub-command, it uses the **GETACCCR** CCC and also subsequently  
7446 drives the Controller Role Handoff Procedure, which uses the I3C Bus Controller Logic and also engages  
7447 the I3C Secondary Controller Logic during the passing of the Controller Role. This procedure is defined in  
7448 the I3C Specification [**MIPI02**] at **Section 5.1.7.2**, and is required for any Host Controller that supports  
7449 Standby Controller mode.

7450 This sub-command's use of field **MIPI\_RESERVED** is:

- 7451
  - Bits[18:12] for sub-field **STBY\_CR\_DYN\_ADDR**
  - Bits[20:19] for sub-field **HANOFF\_RETRY**

7453  
7454

**Table 142 Internal Control Command for Controller Role Handoff Procedure with GETACCCR CCC**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
11 [31:21]	MIPI_RESERVED	W	0x0	MIPI Alliance Reserved area
2 [20:19]	HANDOFF_RETRY	W	0x0	<p><b>Handoff Retry</b>            Indicates whether the I3C Bus Controller Logic should attempt to retry the Controller Role Handoff procedure, if it should fail on the first attempt.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 2'b00: Do not retry on failure</li> <li>• 2'b01: Retry one time, on any failure other than a NACK of the <b>GETACCCR</b> CCC.</li> <li>• All other values are reserved for future use.</li> </ul>
7 [18:12]	STBY_CR_DYN_ADDR	W	0x0	<p><b>Secondary Controller Dynamic Address</b>            Indicates the Dynamic Address of the chosen Secondary Controller (i.e., Controller-capable) Device to which the Controller Role should be passed.</p> <p>The Driver should provide a known valid Dynamic Address of a Controller-capable Device.</p> <p>The Host Controller shall reject any invalid or reserved Addresses (e.g., 7'h7E).</p>
4 [11:8]	MIPI_CMD	W	0x0	<p><b>MIPI Alliance Command</b>  <b>Value:</b> 0x7 for this sub-command</p>
8 [7:0]	<b>Common to all Internal Control Command Sub-command Types</b> (See <b>Table 135</b> )			

7455  
7456 **Table 143** shows the defined values for Response Descriptor field **ERR\_STATUS** and the corresponding meanings.

7457  
7458 **Table 143 Response Descriptor Status for Controller Role Handoff Procedure with GETACCCR CCC**

<b>Value of ERR_STATUS</b>	<b>Condition</b>	<b>Description</b>
0x0	<b>SUCCESS</b>	The Controller Role Handoff Procedure was successful; the chosen Secondary Controller accepted the Controller Role; the Secondary Controller Logic will monitor for Error Type CE3 condition.
0x5	<b>NACK</b>	The chosen Secondary Controller did not ACK its Dynamic Address in Direct CCC framing for the <b>GETACCCR CCC</b> . The Controller Role Handoff Procedure was canceled.
0x8	<b>HC_TERMINATED</b>	Not applicable for this sub-command. (Would only be generated if the Host Controller received any other type of Command Descriptor during the transitional state.)
0xA	<b>NOT_SUPPORTED</b>	When the Host Controller received this sub-command, it was not in the transitional state (i.e., to prepare for the Controller Role Handoff).
0xC	<b>GETACCCR_PARITY</b> <i>(Transfer Type Specific)</i>	The I3C Bus Controller Logic received the correct Dynamic Address, but the Parity Bit in the Secondary Controller's response was incorrect. The Controller Role Handoff Procedure was canceled.
0xD	<b>GETACCCR_DYNADDR</b> <i>(Transfer Type Specific)</i>	The I3C Bus Controller Logic received an incorrect Dynamic Address in the Secondary Controller's response. The Controller Role Handoff Procedure was cancelled.

#### 8.4.2.8 Sub-Command 0xD: Attempt Dead Bus Recovery

7459 **Note:**

7460     *This sub-command is new for this version of the I3C HCI Specification, and is recommended for*  
7461     *Host Controllers that need to detect their role when initializing on an I3C Bus where it is not initially*  
7462     *known if the Bus already has an Active Controller.*

7463 If field **MIPI\_CMD** in the Internal Control Command's Command Descriptor contains 0xD, then the sub-  
7464 command is Attempt Dead Bus Recovery. This sub-command is used by the Driver to engage the Dead Bus  
7465 Recovery Mechanism under the following circumstances:

- 7466     • Before finishing Host Controller initialization steps (i.e., when field **BUS\_ENABLE** in register  
7467       **HC\_CONTROL** is 1'b0), when the presence or current status of an Active Controller on the I3C Bus (i.e.,  
7468       not this Host Controller) is not yet known;

7469     If Standby Controller mode is also supported, then the Driver also shall not write 1'b1 to  
7470       field **ACR\_FSM\_OP\_SELECT** in register **STBY\_CR\_CONTROL** (*Section 7.7.11.1*) before  
7471       engaging the Dead Bus Recovery Mechanism.

- 7472     • While operating in Standby Controller mode (if Secondary Controller Logic is supported), if the Host  
7473       Controller is not the Active Controller, and if the Host suspects that the Active Controller on the I3C Bus  
7474       has stopped responding (or has crashed, been disconnected, or has encountered another condition that  
7475       renders it inoperable).

7476 Additionally, the Host Controller shall always generate a Response Descriptor, and shall return one of  
7477 several error codes in field **ERR\_STATUS**, depending on the results of the sub-command and its attempt to  
7478 detect the state of the I3C Bus and claim the Active Controller role, per *Table 145*. If the Host Controller  
7479 returns 0x0 (**SUCCESS**) in field **ERR\_STATUS** of the Response Descriptor, then this typically indicates a  
7480 successful attempt; however, other interrupt conditions are typically sent due to the conditions on the I3C  
7481 Bus.

7482 This sub-command's use of field **MIPI\_RESERVED** is:

- 7483     • Bit[15] for sub-field **TRY\_STANDBY**
- 7484     • Bits[14:12] for sub-field **DBR\_PROC**

7485

**Table 144 Internal Control Command for Attempt Dead Bus Recovery**

<b>Size [Bits]</b>	<b>Field Name</b>	<b>Memory Access</b>	<b>Reset Value</b>	<b>Description</b>
<b>16 [31:16]</b>	<b>MIPI_RESERVED</b>	W	0x0	MIPI Alliance Reserved area
<b>1 [15]</b>	<b>TRY_STANDBY</b>	W	0x0	<p><b>Try Standby Controller Mode</b>  <b>On HC Initialization only:</b> If the Dead Bus Recovery FSM determines that there is already an Active Controller, then the Host Controller shall engage its Secondary Bus Controller Logic and join the Bus in Standby Controller mode.</p> <p><b>Note:</b>  <i>This requires Secondary Bus Controller Logic that supports Standby Controller mode, as defined in <b>Section 6.17</b>. The Host Controller must support Standby Controller mode as an extended capability. If successful, the Dead Bus Recovery FSM shall engage the Standby Controller mode automatically if it detects an Active Controller on the I3C Bus.</i></p>
<b>3 [14:12]</b>	<b>DBR_PROC</b>	W	0x0	<p><b>Dead Bus Recovery Procedure</b>  Selects the Recovery or Reset procedure.  Refer to the I3C Specification [<b>MIPI02</b>] at <b>Section 5.1.10.2</b> for details on various recovery and reset procedures for an I3C Controller.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li><b>0x0:</b> Only test the I3C Bus: return the status, but do not claim the Active Controller role. If no Active Controller responds, then release SDA.</li> <li><b>0x1:</b> Attempt a simple Error Type DBR procedure that only tests the I3C Bus by pulling SDA Low, but cancels the procedure if it detects an Active Controller.</li> <li><b>0x2:</b> Use the DBR procedure configuration in register <b>DBR_ENGAGE</b> (see <b>Section 7.7.6.1</b>) that can optionally send an IBI, Hot-Join Request or other activity if an Active Controller is detected.</li> <li><b>All other values:</b> Reserved for future use.  See <b>Section 6.18</b> for more information.</li> </ul>
<b>4 [11:8]</b>	<b>MIPI_CMD</b>	W	0x0	<p><b>MIPI Alliance Command</b>  <b>Value:</b> <b>0xD</b> for this sub-command</p>
<b>8 [7:0]</b>	<b>Common to all Internal Control Command Sub-command Types</b> (See <b>Table 135</b> )			

7486  
7487 **Table 145** shows the defined values for Response Descriptor field **ERR\_STATUS** and the corresponding meanings.

7488  
7489 **Table 145 Response Descriptor Status for Controller Role Handoff Procedure with  
GETACCCR CCC**

<b>Value of ERR_STATUS</b>	<b>Condition</b>	<b>Description</b>
0x0	<b>SUCCESS</b>	The Dead Bus Recovery procedure was successful; the requested action in field <b>DBR_PROC</b> was performed. If field <b>DBR_PROC</b> was not 0x02, then this means no other Active Controller was detected on the I3C Bus.
0x5	<b>NACK</b>	The Dead Bus Recovery procedure was attempted, but another Active Controller was detected so the attempt to claim the Controller Role was unsuccessful.
0x8	<b>HC_TERMINATED</b>	The Dead Bus Recovery procedure was cancelled due to a Host Controller error.
0xA	<b>NOT_SUPPORTED</b>	The Internal Control Command was rejected because the Host Controller does not support the Dead Bus Recovery procedure.
0xC	<b>SECONDARY_ROLE (Transfer Type Specific)</b>	If field <b>TRY_STANDBY</b> was 1'b1, then the Secondary Controller Logic was engaged because another Active Controller was on the bus.
0xD	<b>REQUEST_SENT (Transfer Type Specific)</b>	If field <b>DBR_PROC</b> was 0x2, then the requested IBI, Hot-Join, or other request was successfully sent.

#### 8.4.3 Common Aspects of Transfer Commands

The I3C TCRI Specification contains the full normative definition of Transfer Command types (see [MIPI06] at *Section 7.1.2*).

However, a Host Controller shall also observe the following requirements:

- **For Transfer Commands with field MODE = 0x5:** This indicates HDR-Ternary Mode. Selection of HDR-TSP Mode vs. HDR-TSL Mode depends on the value of field **I2C\_DEV\_PRESENT** in register **HC\_CONTROL**. The Host Controller shall use the appropriate Broadcast CCC when entering HDR-Ternary Mode, based on this field.
  - If field **I2C\_DEV\_PRESENT** is 1'b0, then the Host Controller shall use the **ENTHDR1** CCC and enter HDR-TSP Mode. Note that this is only supported for a Pure Bus: no Legacy I<sup>2</sup>C Devices are allowed to be on the I3C Bus when HDR-TSP Mode is used.
  - If field **I2C\_DEV\_PRESENT** is 1'b1, then the Host Controller shall use the **ENTHDR2** CCC and enter HDR-TSL Mode. This HDR Mode may be used if Legacy I<sup>2</sup>C Devices are on the I3C Bus.
- Field **WROC** controls whether Response Status is conditionally required after successful completion of the transfer. The successful completion shall be read from register **RESPONSE\_QUEUE\_PORT**. Upon unsuccessful transfer, the Response Status shall always be sent:
  - **In PIO Mode:** A value of 1'b0 may optionally be used for Write-Type transfers, if the Driver does not wish to receive a Response Descriptor for a successful transfer. However, if there is a transfer error, then the Host Controller shall generate a Response Descriptor regardless of the value in this field. In most cases, the Driver should use a value of 1'b1.
  - **In DMA Mode:** The Host Controller shall always generate a Response Descriptor, and the Driver shall always set this field to 1'b1.
- All Transfer Commands will use a data buffer, based on the currently selected operating mode:
  - **For PIO Mode:** The data buffer is available through Transfer Data Port (RX Data Port and TX Data Port). For Immediate Data Transfer commands, these are defined as Write-type transfers so only the TX Data Port is used. The Driver must write to or read from register **TX\_DATA\_PORT** (see *Section 6.5* and *Section 7.5.3*), and prevent both overflow and underflow conditions from occurring during the transfer.
  - **For DMA Mode:** The buffer is provided in the Transfer Descriptor that contains the Command Descriptor. The Driver must allocate sufficient Host system memory to hold the transfer data (per *Section 6.6*).
  - **For Transfers with No Data Bytes (i.e., zero-length):** These are only intended for special uses, such as CCCs that simply send the Command Code to the Target and then use ACK/NACK alone to determine whether the CCC was successfully received. For most other typical Transfer Commands, software should provide a non-zero value in field **DATA\_LENGTH**.
- Support for Combo Transfer Commands is optional: an implementer may choose whether to implement support for Combo transfers, per the specific use case. If so, then field **COMBO\_COMMAND** in register **HC\_CAPABILITIES** shall have a value of 1'b1 (per *Section 7.4.4*). If not, then field **COMBO\_COMMAND** shall instead have a value of 1'b0.

## 8.5 Response Descriptor

The Response Descriptor is a read-only structure describing the success or failure of a Command, and the amount of data transferred.

This structure is used in two different ways:

- **For PIO Mode:** The Response Descriptor is read from Response Queue (i.e., via reads from Response Queue Port).
- **For DMA Mode:** The Response Descriptor is read from the Response Ring.

The Response Descriptor is 32 bits (i.e., 1 DWORD) in length, and is normatively defined in *Section 7.1.3* of the I3C TCRI Specification [*MIPI06*]. *Table 146* summarizes the relevant fields.

**Table 146 Summary of Response Descriptor Structure Fields**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
4 [31:28]	ERR_STATUS	R	0x0	<b>Response Error Status</b> Indicates the Response status for the processed command (i.e., either success, or the error type encountered). Error codes are described fully in <i>Section 6.13.1</i> of this I3C HCI Specification, and in <i>Section 6.4.1</i> of the I3C TCRI Specification [ <i>MIPI06</i> ]. <b>Values:</b> <ul style="list-style-type: none"><li>• 0x0: <b>SUCCESS</b>: Transfer was successful, no error.</li><li>• 0x1: <b>CRC</b>: CRC Error</li><li>• 0x2: <b>PARITY</b>: Parity Error</li><li>• 0x3: <b>FRAME</b>: Frame Error</li><li>• 0x4: <b>ADDR_HEADER</b>: Address Header Error</li><li>• 0x5: <b>NACK</b>: Address was NACK'ed, or Dynamic Address Assignment was NACK'ed</li><li>• 0x6: <b>OVL</b>: Receive Overflow or Transfer Underflow Error</li><li>• 0x7: <b>I3C_SHORT_READ_ERR</b>: Target returned fewer data bytes than requested in field <b>DATA_LENGTH</b> of a Transfer Command that did not permit a 'short' read (per <i>Section 6.2.7</i> of the I3C TCRI Specification [<i>MIPI06</i>])</li><li>• 0x8: <b>HC_ABORTED</b>: Aborted (i.e., terminated) by Host Controller due to internal error or Abort operation</li><li>• 0x9: Transfer was terminated due to Bus action: either <b>I2C_WR_DATA_NACK</b> (for I<sup>2</sup>C transfers) or <b>BUS_ABORTED</b> (for I3C transfers)</li><li>• 0xA: <b>NOT_SUPPORTED</b>: Command with specific parameters was not supported by the Host Controller implementation (e.g., specific Internal Control codes may not be supported)</li><li>• 0xB: <b>RESERVED</b></li><li>• 0xC–0xF: <b>Transfer Type Specific Errors</b>, defined for specific transfer types</li></ul>

Size [Bits]	Field Name	Memory Access	Reset Value	Description
4 [27:24]	TID	R	0x0	<p><b>Command/Response Transaction ID</b>  Identification tag for the command.  This value shall match the value of field <b>TID</b>, for a previously enqueued Command Descriptor that was sent on the Bus.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• <b>0x0–0xF</b>: Valid Transaction IDs</li> </ul>
8 [23:16]	RESERVED	—	—	—
16 [15:0]	DATA_LENGTH	R	0x0	<p><b>Data Length / Device Count</b>  The meaning of this field depends on the context:  <b>For Write Transfer:</b> Remaining data length (in bytes)  <b>For Read Transfer:</b> Received data length (in bytes)  <b>For Address Assignment:</b> Remaining Device count</p>

## 8.6 IBI Status Descriptor

The IBI Status Descriptor is a read-only structure describing the outcome of I3C Bus activity or other related actions that are **not** the result of the Host Controller processing a normally enqueued Command Descriptor and generating a corresponding Response Descriptor. In general, the IBI Status Descriptor is used for describing IBI Requests and other types of requests that can be raised by I3C Targets, or other events/actions that might be unexpected or autonomously generated within the Host Controller.

**Note:**

*In version 1.1 of this I3C HCI Specification, the IBI Status Descriptor structure was only used to report IBI Requests with data payloads. In this version of the I3C HCI Specification, the structure has been extended to report additional event types.*

**Table 147** defines the fields in the IBI Status Descriptor data structure. Field **STATUS\_TYPE** indicates the source of the status data in the IBI Status Descriptor, and also defines the usage of other fields in the data structure.

The following sub-sections define the uses of the fields of the IBI Status Descriptor, as indicated by the value of field **STATUS\_TYPE**:

• **3'b000: (REGULAR\_IBI):**

- Regular IBI Request notification with data payload, see *Section 8.6.2*
- Notification of Hot-Join Request or Controller-Role Request, see *Section 8.6.3*
- **3'b001: (CREDIT\_ACK):** Target IBI credit update acknowledgement or error, see *Section 8.6.4*
- **3'b010: (SCHEDULED\_CMD):** Scheduled Command execution report, see *Section 8.6.5*
- **3'b100: (AUTOCMD\_READ):** Auto-Command read notification with data payload, see *Section 8.6.6*
- **3'b111: (STBY\_BR\_BCAST\_CCC):** Broadcast CCCs received in Standby Controller mode, see *Section 8.6.7*

**Table 147 IBI Status Descriptor Structure**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
1 [31]	<b>IBI_STS</b> <i>(Extended in I3C HCI v1.2)</i>	R	0x0	<b>IBI Received Status</b> Indicates how the event or action was handled. Specific usage depends on value of field <b>STATUS_TYPE</b> .
1 [30]	<b>ERROR</b> <i>(Extended in I3C HCI v1.2)</i>	R	0x0	<b>Error</b> Indicates whether an error was encountered. The following errors shall be detected: <ul style="list-style-type: none"><li>• CRC Error</li><li>• Parity Error</li><li>• Target Address NACK</li><li>• Broadcast Address (7'h7E) NACK</li></ul> Specific usage depends on value of field <b>STATUS_TYPE</b> .

Size [Bits]	Field Name	Memory Access	Reset Value	Description
3 [29:27]	<b>STATUS_TYPE</b> <i>(Expanded in I3C HCI v1.2 – see Note after this Table)</i>	R	0x0	<p><b>Status Type</b> Indicates the source of the status data and determines the meaning of fields <b>IBI_STS</b> and <b>ERROR</b>. Specific usage is noted in sub-sections below.</p> <p><b>Values:</b></p> <ul style="list-style-type: none"> <li>• 3'b000: <b>REGULAR_IBI</b>: Regular IBI received</li> <li>• 3'b001: <b>CREDIT_ACK</b>: Target IBI credit counter update acknowledgement (per <b>Section 6.9.5</b>)</li> <li>• 3'b010: <b>SCHEDULED_CMD</b>: Generated by processing Scheduled Commands (per <b>Section 6.16</b>)</li> <li>• 3'b100: <b>AUTOCMD_READ</b>: Data from subsequent Auto-Command initiated after receiving IBI with matching MDB (per <b>Section 6.11</b>)</li> <li>• 3'b111: <b>STBY_CR_BCAST_CCC</b>: Data from certain Broadcast CCCs sent by the Active Controller (per <b>Section 6.17.3.2</b>)</li> <li>• <b>All Other Values</b>: Reserved.</li> </ul>
1 [26]	RESERVED <i>(Changed in I3C HCI v1.2 – see Note after this Table)</i>	—	—	—
1 [25]	<b>TS</b>	R	0x0	<p><b>IBI Timestamp Present</b> Indicates whether a Timestamp is available for the IBI. Specific usage is noted in sub-sections below.</p>
1 [24]	<b>LAST_STATUS</b>	R	0x0	<p><b>Last IBI Status</b> Last IBI status for the IBI transaction.</p> <p><b>Note:</b> <i>Even if LAST_STATUS is set to 1'b0, the Driver software shall still evaluate the data payload length by examining the CHUNKS (DMA Mode) or DATA_LENGTH (PIO Mode) field.</i></p>
8 [23:16]	<b>CHUNKS</b>	R	0x0	<p><b>IBI Valid Chunks Count</b> Number of valid data Chunks in the IBI Data Ring (DMA Mode) that are associated with this IBI Status Descriptor.</p> <p><b>Note:</b> <i>This field is not used in PIO Mode.</i></p>
8 [15:8]	<b>IBI_ID</b>	R	0x0	<b>IBI Received ID</b> The meaning of this field depends on the context. Usage depends on value of field <b>STATUS_TYPE</b> .

Size [Bits]	Field Name	Memory Access	Reset Value	Description
8 [7:0]	DATA_LENGTH	R	0x0	<b>IBI Data Length</b> The meaning of this field depends on the context. Usage depends on value of field <b>STATUS_TYPE</b> . <b>For PIO Mode:</b> Number of data bytes in IBI Data. <b>For DMA Mode:</b> Number of data bytes in last Data Chunk.

7560      **Note:**

7561      In version 1.1 of this I3C HCI Specification, field **HW\_CONTEXT** (i.e., Bits[28:26]) was defined as a  
7562      3-bit field that allowed for the I3C Bus Controller Logic to describe hardware-specific context for IBI  
7563      processing. Implementers had the option to use this field to provide more information to the SW  
7564      driver. However, actual use of this field proved to be limited, and implementers did not define a  
7565      standard encoding for it. In this version of the I3C HCI Specification, field **HW\_CONTEXT** has been  
7566      removed since Bits[28:27] are now allocated to an expanded **STATUS\_TYPE** field. This allows  
7567      additional reporting of events from other sources beyond just IBI Requests. Considering  
7568      Bits[29:27], the existing values with Bits[28:27] set to 2'b00 have the same meaning as previous  
7569      versions of the HCI Specification.

### 8.6.1 Common Usage

- The following fields in the IBI Status Descriptor shall be used commonly across multiple types of events:
- Field **LAST\_STATUS** shall indicate whether this is the last IBI Status Descriptor for the event.
    - In cases where a data payload requires multiple chunks, all chunks except the last shall have **LAST\_STATUS** set to 1'b0, and the last chunk shall have **LAST\_STATUS** set to 1'b1.
  - Field **CHUNKS**:
    - In DMA Mode, this field shall indicate the number of valid data Chunks in the IBI Ring that are associated with this IBI Status Descriptor.
    - In PIO Mode, this field shall not be used.
  - Field **DATA\_LENGTH** shall describe the length of data enqueued to the IBI Queue/Ring that is associated with this IBI Status Descriptor:
    - In PIO Mode, this field shall contain the number of bytes for this chunk.
    - In DMA Mode, this field shall contain the number of bytes in the last Data Chunk only. If there are any preceding Data Chunks, then they shall have the maximum number of data bytes.

### 8.6.2 Usage for Regular IBIs

The Host Controller shall generate IBI Status Descriptors for IBI Requests that are received from I3C Targets on the Bus. Such IBI Status Descriptors shall have field **STATUS\_TYPE** set to 3'b000 (**REGULAR\_IBI**), and the other fields in the structure shall be used as follows:

- Field **IBI\_STS** shall indicate whether the IBI Request was ACKed or NACKed by the Host Controller:
  - If the Host Controller ACKed the IBI Request, then field **IBI\_STS** shall be set to 1'b0.
  - If the Host Controller NACKed the IBI Request, then field **IBI\_STS** shall be set to 1'b1.
- Field **ERROR**:
  - Shall be set to 1'b0 (**NORMAL**) if the IBI data payload ended normally (i.e., was not terminated by the Host Controller), or
  - Shall be set to 1'b1 (**TERMINATED**) if the Host Controller terminated the IBI data payload for any of the following reasons:
    - The Host Controller shall terminate the IBI data payload if the IBI Queue/Rings become full and cannot accept any new IBI Status Descriptors or associated data DWORDs.
    - If the Target IBI credit counting mechanism is supported and enabled (per **Section 6.9.5**), then the Host Controller shall terminate the IBI data payload if there is insufficient credit for that Target during an IBI payload (per the value of field **TERM\_READ\_ZERO\_CREDIT**).
    - If the Host Controller is using coalesced data reporting for the Auto-Command read transfer after the IBI data payload (per **Section 6.11.3**), then the Host Controller shall terminate the read transfer if it encounters any of the following errors during the Auto-Command read transfer:
      - CRC Error or Parity Error (for HDR Modes only)
      - Target NACKs its Dynamic Address
      - Broadcast Address (7'h7E) NACK
- Field **TS** shall be set to 1'b1 if a Timestamp is available for the IBI (per **Section 6.9.3**) as part of the data payload, or 1'b0 if no Timestamp is available for the IBI.
- For field **IBI\_ID**, Bits[15:9] shall contain the Target's Dynamic Address, and Bit[8] shall contain the RnW bit (i.e., RnW = 1 for a regular IBI).
- Fields **DATA\_LENGTH** and **CHUNKS** shall be used to describe the length of the read data for this IBI Status Descriptor (per **Section 8.6.1**).

If the IBI has a data payload, then the Host Controller shall enqueue the data payload bytes in one or more DWORDs per the current operating mode, using chunks (per **Section 8.6.1**).

### 8.6.3 Usage for Hot-Join Requests and Controller-Role Requests

The Host Controller shall generate IBI Status Descriptors for either Hot-Join Requests or Controller-Role Requests that are received from I3C Targets on the Bus. Such IBI Status Descriptors shall have field **STATUS\_TYPE** set to 3'b000 (**REGULAR\_IBI**), and the other fields in the structure shall be used as follows:

- Field **IBI\_STS** shall indicate whether the request was ACKed or NACKed by the Host Controller:
  - If the Host Controller ACKed the request, then field **IBI\_STS** shall be set to 1'b0.
  - If the Host Controller NACKed the request, then field **IBI\_STS** shall be set to 1'b1.
- Fields **ERROR**, **TS**, **LAST\_STATUS**, and **DATA\_LENGTH** shall be set to 1'b0 always.
- Field **IBI\_ID**:
  - For Hot-Join Requests, Bits[15:9] of this field shall contain the Hot-Join ID (i.e., 7'h02 always), and Bit[8] of this field shall contain 1'b0 (i.e., RnW = 0)
  - For Controller-Role Requests, Bits[15:9] of this field shall contain the Secondary Controller's Dynamic Address, and Bit[8] of this field shall contain the RnW bit (i.e., RnW = 0 for a Controller-Role Request).

**Note:**

*Bit[8] (i.e., the RnW bit) differentiates these request types from a regular IBI Request. Additionally, these request types do not have a data payload.*

### 8.6.4 Usage for Target IBI Credit Counter Updates

If the Target IBI credit counting mechanism is supported and enabled (per [Section 6.9.5](#)), then the Host Controller shall generate IBI Status Descriptors to report the processing of Target IBI credit update requests. Such IBI Status descriptors shall have field **STATUS\_TYPE** set to 3'b001 (**CREDIT\_ACK**).

If such a request is successfully processed, then the other fields in the structure shall be used as follows:

- Field **IBI\_STS** shall be set to 1'b0 (**UPDATED**) to report a successful update.
- Field **LAST\_STATUS** shall be set to 1'b0.
- For field **IBI\_ID**, Bits[15:9] shall be set to the Target's Dynamic Address, and Bit[8] shall be set to 1'b0.
- Field **DATA\_LENGTH** shall be set to 4 (i.e., a single DWORD).
- If DMA Mode is used, then field **CHUNKS** shall report the number of data chunks in this report (i.e., 1 chunk for 1 DWORD of data).
- All other fields shall be set to zero.

The data payload for this IBI Status Descriptor shall be a single DWORD that contains the updated credit counter for the IBI Target in Bits[15:0], and zero in Bits[31:16].

- If PIO Mode is used, then this DWORD shall be enqueued into the IBI Queue after this IBI Status Descriptor (per [Section 6.9.1](#)).
- If DMA Mode is used, then this DWORD shall be enqueued into the corresponding IBI Data Ring (per [Section 6.9.2](#)).

If such a request is not successfully processed (i.e., dropped by the Host Controller), then the other fields in the structure shall be used as follows:

- Field **IBI\_STS** shall be set to 1'b1 (**DROPPED**) to report an unsuccessful update.
- Field **LAST\_STATUS** shall be set to 1'b0.
- For field **IBI\_ID**, Bits[15:9] shall be set to the Target's Dynamic Address, and Bit[8] shall be set to 1'b0.
- Field **DATA\_LENGTH** shall be set to 0 (i.e., no data).
- All other fields shall be set to zero.

### 8.6.5 Usage for Scheduled Command Execution Reports

If the Scheduled Commands logic is supported and enabled (per *Section 6.16*) then the Host Controller shall generate IBI Status Descriptors to report the execution of Transfer Commands by the Scheduled Commands logic. Such IBI Status Descriptors shall have field **STATUS\_TYPE** set to 3'b010 (**SCHEDULED\_CMD**), and the other fields in the structure shall be used as follows:

- Field **IBI\_STS** shall be set to 1'b0 always.
- Field **ERROR**:
  - Shall be set to 1'b0 (**NO\_ERROR**) if the Transfer Command was executed normally (i.e., if no errors were detected by the Host Controller), or
  - Shall be set to 1'b1 (**ERROR**) if the Host Controller detected an error during Transfer Command execution.
- Field **IBI\_ID** shall identify the context, which depends on the Handler Type of the specific Scheduled Commands instance:
  - If the Transfer Command was processed by Handler Type 1, then see *Section 7.7.8.6*.
  - If the Transfer Command was processed by Handler Type 2, then see *Section 7.7.9.7*.
  - If the Transfer Command was processed by Handler Type 3, then see *Section 7.7.10.9*.
- Fields **DATA\_LENGTH** and **CHUNKS** shall be used to describe the length of the read data for this IBI Status Descriptor (per *Section 8.6.1*).

If the Transfer Command was a Read-Type transfer, then the Host Controller shall enqueue the read data payload bytes in one or more DWORDs per the current operating mode, using chunks (per *Section 8.6.1*).

### 8.6.6 Usage for Auto-Command Read Data

If Auto-Command is supported and enabled, and if the Host Controller supports separated reporting of Auto-Command read data (per *Section 6.11.3*), then the Host Controller shall generate IBI Status Descriptors to contain data from an Auto-Command read transfer from a Target Device on the I3C Bus.

Such IBI Status Descriptors shall have field **STATUS\_TYPE** set to 3'b100 (**AUTOCMD\_READ**). For Auto-Command read, the Target would have previously initiated an IBI Request with a matching Mandatory Data Byte (MDB) to indicate a Pending Read; as a result, the first instance of an IBI Status Descriptor of type **AUTOCMD\_READ** shall always follow one or more IBI Status Descriptors of type **REGULAR\_IBI** (since the IBI data payload and Auto-Command data payload would be reported separately).

The other fields in such an IBI Status Descriptor structure shall be used as follows:

- Field **IBI\_STS** shall indicate whether the Auto-Command read was ACKed or NACKed by the Target that raised the read transfer (i.e., the Pending Read IBI that triggered the Host Controller to perform the Auto-Command read):
  - If the Target ACKed the read transfer, then field **IBI\_STS** shall be set to 1'b0.
  - If the Target NACKed the read transfer, then field **IBI\_STS** shall be set to 1'b1.
- Field **ERROR**:
  - Shall be set to 1'b0 (**NORMAL**) if the Auto-Command read data payload ended normally (i.e., was not terminated by the Host Controller), or
  - Shall be set to 1'b1 (**TERMINATED**) if the Host Controller terminated the Auto-Command read data payload, for any of the following reasons:
    - The Host Controller shall terminate the Auto-Command read transfer if the IBI Queue/Rings become full and cannot accept any new IBI Status Descriptors or associated data DWORDs.
    - If the Target IBI credit counting mechanism is supported and enabled (per *Section 6.9.5*) then the Host Controller shall terminate the Auto-Command read data payload if there is insufficient credit for that Target during an IBI payload (per the value of field **TERM\_READ\_ZERO\_CREDIT**).
    - The Host Controller shall terminate the Auto-Command read transfer if it encounters any of the following errors during the Auto-Command read transfer:
      - CRC Error or Parity Error (for HDR Modes only)
      - Broadcast Address (7'h7E) NACK
- Field **TS** shall be set to 1'b0 always.
- For field **IBI\_ID**, Bits[15:9] shall contain the Target's Dynamic Address, and Bit[8] shall contain the RnW bit (i.e., RnW = 1 for an Auto-Command read).
- Fields **DATA\_LENGTH** and **CHUNKS** shall be used to describe the length of the read data for this IBI Status Descriptor (per *Section 8.6.1*).

If the Auto-Command read has a data payload (i.e., is not a zero-length read transfer), then the Host Controller shall enqueue the data payload bytes in one or more DWORDs per the current operating mode, using chunks (per *Section 8.6.1*).

### 8.6.7 Usage for Broadcast CCCs Received in Standby Controller Mode

If the Host Controller supports Standby Controller mode and is currently operating as a Secondary Controller (i.e., not the Active Controller), then the current Active Controller of the I3C Bus may send Broadcast CCCs that will be received by the Host Controller. The Host Controller shall generate IBI Status Descriptors to contain data from such Broadcast CCCs from the Active Controller of the I3C Bus (per [Section 6.17.3.2](#)). Such IBI Status Descriptors shall have field **STATUS\_TYPE** set to 3'b111 (**STBY\_CR\_BCAST\_CCC**), and the other fields in the structure shall be used as follows:

- Fields **IBI\_STS**, **ERROR**, and **TS** shall be set to 1'b0 always.
- For field **IBI\_ID**, Bits[15:9] shall be set to the Broadcast Address (7'h7E), and Bit[8] shall be set to 1'b0.
- Fields **DATA\_LENGTH** and **CHUNKS** shall be used to describe the length of the read data for this IBI Status Descriptor (per [Section 8.6.1](#)).

The Host Controller shall also capture the CCC data payload (see [Section 6.17.3.2](#)) and enqueue the data payload bytes in one or more DWORDs per the current operating mode, using chunks (per [Section 8.6.1](#)).

## 8.7 Memory Descriptor

**Note:**

This section applies only for Host Controller implementations that connect to a System Bus that supports memory-mapped IO and DMA, and that implement any of the following: (A) DMA Mode, (B) Device Context tables in Host system memory, or (C) Scheduled Commands logic that uses Handler Type 3.

The Memory Descriptor is a structure used to describe a single block of physical memory. An array of Memory Descriptors is used to describe a Scatter-Gather Buffer, which is a Data Buffer defined with multiple blocks of physically contiguous memory.

**Table 148 Memory Descriptor Structure**

Size [Bits]	Field Name	Memory Access	Reset Value	Description
32 [95:64]	<b>BUFFER_PTR_HI</b>	R/W	0x0	<b>Buffer Pointer High</b>
21 [63:32]	<b>BUFFER_PTR_LO</b>	R/W	0x0	<b>Buffer Pointer Low</b>
16 [31:16]	RESERVED	R/W	0x0	–
16 [15:0]	<b>BLOCK_SIZE</b>	R/W	0x0	<b>Buffer Block Size</b> Buffer size in Bytes

This page intentionally left blank.

## 9 System Bus Implementations

This Section describes the requirements and details of Host Controllers that implement connection to various types of System Buses.

### 9.1 System Bus Requirements

The generic Host Controller Interface definition (listed in earlier Sections of this Specification) is applied in accordance to the features available in a particular implementation's System Bus. Some I3C HCI features do not map well to all System Bus types, so the capabilities available in a particular System Bus may dictate which features in a Host Controller can be implemented, and which ones cannot be supported.

The following System Bus features are prerequisites for any Host Controller implementation:

- A method for the Host to discover the Host Controller on the System Bus, configure its System Bus interface logic for the Host to access, and enumerate its capabilities correctly
- A method for the Host to access the Host Controller registers, using read/write commands with a register offset

The following System Bus features are optional, but their presence or absence may determine which features or capabilities of a Host Controller are supported (or even chosen to be implemented), as well as the method for performing transfers, receiving interrupts including IBI Status notifications, or performing advanced operations to manage the Host Controller:

- Direct memory access (or similar) capability that allows the Host Controller to access Host system memory.
  - This capability allows the Device to use a Memory Access Engine (i.e., a DMA engine) to request access to Host system memory, without direct software involvement from the Host. The Device also has a Memory Access Interface (see **Figure 3**) to manage internal requests to specific memory addresses for defined purposes, each configured by the Driver with regions of allocated system memory.
  - Such a capability requires corresponding data access logic within the Host system's System Bus controller logic, or other entities that might serve as a "Hub", "Router", and/or "Switch" between the System Bus controller logic and the Host Controller (i.e., things necessary to connect the Device to a specific Port on the System Bus) to respond to such requests initiated by a Device, and fulfill them on behalf of the Host System's memory controller(s).
- A method for the Host to poll for interrupt notifications, or be proactively notified by the Host Controller when an interrupt notification condition is satisfied
- A method for the Host to send requests, according to the capabilities of the System Bus interface specification, to access certain features in the Host Controller
- Support for one or more Host Controller Interface instances within the same Host Controller may be exposed:
  - Either within a single interface, as a single mapping of HCI registers with a Multi-Bus Instance extended capability
  - Or as separate interfaces, i.e., multiple independent mappings of HCI registers using System-Bus-specific methods to direct requests to different interfaces

## 9.2 Implementation Specifics

7766

This Section lists the implementation details for common types of System Buses.

### 9.2.1 PCI and PCI-Compatible Buses

7767  
7768  
7769  
7770

Although PCI devices have largely been supplanted by PCI-Express devices, the higher-level software compatibility with PCI has been largely preserved. Therefore this Specification will not cover the lower-level electrical details, but will focus instead on the logical details from a software perspective. The term “PCI\*” will refer to a System Bus that maintains logical compatibility with PCI.

7771  
7772  
7773  
7774  
7775  
7776

PCI\* Devices generally support memory-mapped IO (MMIO) from the Host system address map, and enable interrupt Device-initiated interrupt notifications via one of several methods: Legacy (INTx), MSI, or MSI-X. This enables a basic method of interacting with a PCI\* I3C Host Controller, for Host Controller implementations that implement PIO Mode by the use of basic MMIO requests comprising simple register read/write accesses as smaller packets initiated via Host transactions, to produce entries in the Command Queue and read Response Queue entries and IBI Notifications directly.

7777  
7778  
7779  
7780  
7781  
7782  
7783

In most System Bus implementations, PCI\* Devices also have the ability to access Host system memory via “upstream” read and write requests using Bus Mastering (this is the feature name as used by PCI SIG). A PCI\* I3C Host Controller might be implemented with DMA engines, in order initiate read/write requests to Host System memory without software involvement. The combination of these advanced capabilities also enables a higher-performance method of interacting with a PCI\* I3C Host Controller, for Host Controller implementations that implement DMA Mode by the use of upstream memory accesses and also possess a Ring Controller, which can enable pipelined processing of transaction streams.

7784  
7785

Depending on the choices made by an implementer and the System Bus designer, a PCI\* I3C Host Controller may therefore implement DMA Mode and/or PIO Mode.

7786

#### For all PCI\* I3C Host Controller options:

7787  
7788  
7789  
7790  
7791

- Host Controller configuration and enumeration uses a PCI Configuration Space, including PCI Configuration Registers.
- HCI register read/write access is enabled using memory-read and memory-write requests. As the System Bus supports byte-enables, most registers will most likely be accessed by Byte, Word (i.e., 16-bit), or DWORD (i.e., 32-bit) transactions.
  - For specific registers that act as mailbox registers to a Queue, FIFO, or Buffer (e.g., the PIO Queue registers in *Section 7.5*) and have special actions that either produce or consume entries on these structures, the Host Controller shall only allow DWORD transactions. The byte-enables for such transactions must select all 4 bytes comprising the 32-bit mailbox register. Any transactions not selecting all 4 bytes shall not be supported, and should be dropped or rejected as an unsupported request.
  - The DAT and DCT may be exposed either within the HCI register map, or stored in Host system memory (allocated by the Driver) that requires DMA capability within the Host Controller (i.e., the Memory Access Engine).
  - Interrupt notifications are sent via a supported PCI\* interrupt method.

7800  
7801  
7802  
7803

#### For a PCI\* I3C Host Controller that supports PIO Mode:

- The PIO Mode registers are also implemented.

7804 **For a PCI\* I3C Host Controller that supports DMA Mode:**

- 7805 • If PIO Mode is also supported, then PIO registers must be blocked from register read/write access, if the  
7806 Device is configured to use DMA Mode.  
7807 • The Ring Headers Registers are also implemented, with at least one Ring Bundle.  
7808 • A Ring Controller is implemented, and uses the DMA capability within the Host Controller (i.e., the  
7809 Memory Access Engine) to automatically manage transfers enqueued via Ring Bundles that are enabled  
7810 and running.  
7811 • The Memory Access Interface automatically initiates read and write requests to Host system memory,  
7812 to appropriate memory addresses provided in Transfer Descriptors, using the Memory Access  
7813 Interface.

7814 A PCI\* I3C Host Controller may be the sole “function” or Interface within a PCI\* Device, or it may be one  
7815 of many Interfaces within a PCI\* Device. If the I3C Host Controller is part of a multi-function Device, then  
7816 the implementer shall appropriately define the Device’s interface logic to use separate PCI Bus, Device,  
7817 and/or Function numbers for each “function” to ensure that the I3C Host Controller appears as a unique  
7818 “function” so that it can be bound to its own software Driver.

7819 For PCI-Express (“PCIe”), specific other implementation details may apply, such as the configuration of  
7820 the PCIe Root Port, or any other PCIe Switch devices that might be used to connect the I3C Host Controller  
7821 as a Device to the PCIe Root Port.

### 9.2.1.1 PCI\* Configuration Registers

7822 Per PCI SIG [*PCISIG01*], compliance to this MIPI I3C Host Controller Interface Specification is indicated  
7823 in the PCI header with Base Class equal to 0xCh, Sub Class equal to 0xAh, and Programming Interface  
7824 equal to 0x0h.

7825 The Device’s System Bus interface logic shall enable decode to enable read/write access the HCI registers  
7826 defined in the Register Map section, via memory-mapped BAR0. The PCI configuration registers shall  
7827 correctly enable such MMIO access, once the System Bus interface logic is configured to enable Bus  
7828 Mastering (this is the feature name as used by PCI SIG) and MMIO access. The System Bus interface logic  
7829 shall ensure that the Host Controller is correctly enumerated by the Host system’s BIOS or Firmware, in  
7830 order to assign a valid system address range to BAR0 and properly configure any upstream PCI/PCIe  
7831 compliant root ports, bridges or switches to steer access to the assigned system address range to this  
7832 Device.

7833 The Host Controller may implement support for either 32-bit or 64-bit addressing at the discretion of the  
7834 implementer. The Host Controller shall expose all HCI registers in this memory-mapped BAR, starting at  
7835 the initial offset (0x0) for the first I3C Bus Controller instance. The register offsets for all HCI registers  
7836 must fit within the BAR0 size specified in the PCI configuration.

### 9.2.2 Universal Serial Bus (USB)

An I3C Host Controller may also be implemented as a USB Device, conforming to USB 2.0 [*USB01*], USB 3.2 [*USB02*], or beyond.

The details of exposing an I3C Host Controller as a USB Device are defined in the USB-IF's ***USB I3C Device Class Specification [USB03]*** (see *Section 9.2.2.1*).

USB Devices do not have the ability to directly access Host system memory, and generally cannot initiate any such transactions from the Device. Since the Host's USB Host Controller (e.g., EHCI, xHCI) directs the flow of all Bus operations, a USB I3C Host Controller must rely on Host-initiated requests to drive all transactions, including IBI Status notifications.

USB defines the use of a default Control endpoint (EP #0) that supports both standard request types, "class-specific" request types, and "vendor-specific" request types. USB also defines Interrupt endpoints, which are used by USB Host Controllers and their drivers to automate the delivery of interrupt notifications while still working within the Host-initiated traffic model.

The combination of these System Bus capabilities enables a basic method of interacting with a USB I3C Host Controller, for Host Controller implementations that either (A) implement PIO Mode through the use of USB requests that comprise simple register read/write accesses, or (B) provide another mechanism for interacting with the Host Controller that does not directly use PIO Mode.

USB also supports the use of Bulk Transfer endpoints, which allow for packets with large data payloads (up to 512 bytes for USB 2.0 "High-Speed", or 1024 bytes for USB 3.x "SuperSpeed" and its derivatives). In this case the USB device might provide a method to send large USB Bulk packets to PIO Mode queues, or it might enable a higher-level transaction-based command/response flow (i.e., another operating mode) that uses USB Bulk packets for performance, together with other USB protocol-specific packets for efficient flow control.

#### For All USB I3C Host Controller Options:

- Device configuration and enumeration may use USB descriptors, including at least one I3C-specific descriptor that exposes the I3C Interface details and Host Controller configuration.
- HCI register read/write access may be enabled using USB request types via an endpoint.
- The DAT and DCT may be exposed either within the HCI register map, or else stored within an internal structure (local context) that is accessed solely using USB requests. These requests might use an endpoint created specifically for DAT and DCT access.
- Interrupt notifications may be queued within the Device, and delivered via an Interrupt endpoint that is regularly polled by the Host.

DMA Mode (i.e., accessing Host system memory in order to initiate an I3C transfer) is not supported, since the necessary features to enable Direct Memory Access by a Device are not supported by USB. Note that DMA engines might be used internally to access USB Device local memory (i.e., cache within the USB device), however this shall not be presented to the USB Host as "DMA Mode" as defined by this I3C HCI Specification.

For all the descriptors, request types, data structures, and flows listed in this Specification, an implementer may either define their own ("vendor-specific") specification, or else conform to the "Device Class" specification (USB I3C Device Class Specification [*USB03*]). The Device Class Specification refers to this I3C HCI Specification, and provides additional details that are specific to a conforming USB I3C Host Controller.

A USB I3C Host Controller might be the sole "function" or Interface within a USB Device, or it might be one of many Interfaces within a USB "Composite Device". If the I3C Host Controller is part of such a multi-function Composite Device, then the implementer should use the appropriate conventions from the USB device framework and descriptor hierarchy, to ensure that each Interface can be found by its correct Driver.

### 9.2.2.1 USB Descriptors and Endpoints (Conforming to Device Class)

A USB I3C Host Controller that conforms to the Device Class Specification indicates this via its Device Descriptor, or for a multi-function Composite Device, via its Interface Descriptor, by using the standard Class Code and SubClass Code values defined in that Device Class specification. Per the USB I3C Device Class Specification [*USB03*], a conforming USB I3C Host Controller shall have an Interface Descriptor with a Class Code equal to 0x3C.

The USB I3C Host Controller contains an Interface Descriptor to contain all the USB endpoints associated with a given Bus Controller instance. Each Interface Descriptor might also have other “class-specific” accompanying descriptors that describe each Interface in more detail and enumerate the capabilities of an Interface’s supported operating modes or other parameters.

For a USB Device that implements multiple Bus Controller instances, each instance might have its own Interface Descriptor for an I3C Interface, with separate endpoints for Interrupts and Bulk Transfer. Alternately, some implementations with multiple Bus Controller instances might use shared Interrupt endpoints for consolidated delivery of interrupt notifications to the Host. The use of multiple Interface Descriptors per USB Device replaces the use of the Multi-Bus Instance Extended Capability structure (see *Section 7.7.4*), which is not recommended or supported for use by the Device Class specification.

The Device Class Specification defines the particular “mapping” used by the USB function logic within the Device, which allows the Host to enqueue sequences of one or more I3C Transfer Commands using packets sent via the Bulk Transfer “outbound” endpoint. The USB function logic uses the I3C Bus Controller Logic to drive the Transfer Commands on the I3C Bus, and then return the responses to the Host, using packets sent via the Bulk Transfer “inbound” endpoint. The Device Class specification defines the specific framing and data structure formats for such USB Bulk packets that contain I3C Transfer Commands and responses, as well as I3C In-Band Interrupt notifications.

The Device Class Specification also defines “class-specific” request types that are supported by the Device’s default Control endpoint (EP #0) to enable or disable certain Host Controller features, access other data structures or perform commands for initialization and configuration.

The USB endpoint logic within the Device shall steer all class-specific requests to the correct Interface using the “wIndex” field.

### 9.2.3 AMBA Interfaces

An implementation of an I3C Host Controller may use any of the publicly available Arm AMBA specifications (Advanced Microcontroller Bus Architecture), to implement I3C Bus Controller functionality as a peripheral of an on-chip interconnect (i.e., a block that is integrated into the SoC).

I3C Host Controller implementations should use either of the following AMBA standards:

- AMBA 5 AHB standard (i.e., as defined in the AMBA 5 AHB specification)
- AMBA 5 APB standard (i.e., as defined in the AMBA 5 APB specification)

A Host Controller that complies with either of these AMBA standards typically includes a subordinate-type interface (i.e., a target) that can be addressed by one or more manager-type interfaces (i.e., initiators) for HCI register access, including the PIO Queue Port registers (if PIO Mode is supported). Additionally, if such a Host Controller implements DMA Mode and/or has access to Host system memory on the same AMBA interface (i.e., for Device Context) then it would also use a DMA engine (i.e., Memory Access Engine). For operation in DMA Mode, the Memory Access Interface logic would drive memory read/write transactions using the DMA engine, which might be external to the Host Controller block.

## Annex A Implementation Guidance (Informative)

### A.1 Relationship to the I3C Transfer Command/Response Interface (TCRI) Specification

A Host Controller as defined in this I3C HCI Specification contains an implementation of I3C Bus Controller Logic, and also supports the Transfer Command/Response Interface as defined in the I3C TCRI Specification (see [*MIPI06*]). The remainder of the normative definitions in this I3C HCI Specification comprise the Application (as that term is defined in the I3C TCRI Specification), along with the Driver or other upper-level software that would run on the connected Host system and enqueue Transfer Commands to be processed.

The I3C TCRI Specification defines the requirements for processing Transfer Commands and Transfer Responses, whether issued individually or in continuous sequences. The mechanisms for enqueueing and dequeuing such Transfer Commands and Transfer Responses (i.e., the HCI-defined operating modes) are specific to this I3C HCI Specification, including the use of a Transfer Data Port (for PIO Mode) or Host system memory accessed via a Transfer Descriptor (for DMA Mode).

To understand the full set of normative requirements for a Host Controller and its I3C Bus Controller Logic, implementers of a Host Controller should read this I3C HCI Specification in conjunction with the I3C TCRI Specification. Where possible, this I3C HCI Specification contains references to the particular relevant sections of the I3C TCRI Specification. However, a full reading of the I3C TCRI Specification is strongly recommended.

In addition, the I3C Bus Controller Logic shall also comply with the electrical and protocol requirements in version 1.1.1+ of the I3C Specification (see [*MIPI05*]). Implementers are strongly recommended to always use the latest I3C Specification version, including any future updates or Errata that MIPI Alliance might publish.

### A.2 Variable Data Rates and Queue/FIFO Fill Levels

An implementer might choose to dynamically adjust the clock rates or effective data transfer rates, within the limits set by the I3C Specification, in order to slow down the rate of data transmission during a transfer and prevent underrun or overrun conditions for an internal Queue, Tx Data Buffer, or Rx Data Buffer. This might be an advisable capability for PIO Mode, which would help to mitigate issues where the Driver allowed such a Queue or Buffer to approach an underrun or overrun condition due to system latency.

If such a capability is needed, then the implementer should provide control over this capability to software, exposed as registers in a vendor-defined Extended Capability structure. At minimum, such an optional capability should include an enable/disable control, as well as threshold values for “nearly full” and “nearly empty” (or similar) to set the points at which the I3C Bus Controller Logic should start ramping down from its current transfer rate to a lower rate, in order to avoid encountering the overrun/underrun condition at the programmed rate.

This page intentionally left blank.

## Participants

The list below includes those persons who participated in the Working Group that developed this Specification and who consented to appear on this list.

Guruprasad Ardhanari, Intel Corporation  
Mohammad Asad Javed, Intel Corporation  
Noah Bacon, Intel Corporation  
Rajesh Bhaskar, Intel Corporation  
Anamitra Chakrabarti, Synopsys, Inc.  
Rob Gough, Intel Corporation  
Chris Grigg, MIPI Alliance (Team)  
Takayuki Hirama, Sony Group Corporation  
Paul Kimelman, NXP Semiconductors  
Makoto Nariya, Sony Group Corporation  
Pratap Neelashetty, Synopsys, Inc.  
Laura Nixon, MIPI Alliance (Team)

Past Contributors to v1.1:

Rajesh Bhaskar, Intel Corporation  
Anamitra Chakrabarti, Synopsys, Inc.  
Ken Foust, Intel Corporation  
Rob Gough, Intel Corporation  
Chris Grigg, MIPI Alliance (Team)  
Paul Kimelman, NXP Semiconductors  
Pratap Neelashetty, Synopsys, Inc.  
Laura Nixon, MIPI Alliance (Team)

Tomasz Pielaszkiewicz, Intel Corporation  
Radu Pitigoi-Aron, Qualcomm  
Nicolas Pitre, BayLibre  
Guruprasad Ramachandra, Synopsys, Inc.  
Rob Santoro, MIPI Alliance (Team)  
Matthew Schnoor, Intel Corporation  
Katherine Valenti, MIPI Alliance (Team)  
Suresh Venkatachalam, Synopsys, Inc.  
Kasper Wszolek, Intel Corporation  
Qijie Yang, Intel Corporation  
Tadaaki Yuba, Sony Group Corporation  
Fred Zhou, Intel Corporation

Tomasz Pielaszkiewicz, Intel Corporation  
Nicolas (Nico) Pitre, Bay Libre, Inc.  
Guruprasad (T R) Ramachandra, Synopsys, Inc.  
Matthew Schnoor, Intel Corporation  
Seong (Seongmin) Shim, Samsung Electronics, Co.  
Vitor Soares, Synopsys, Inc.  
Eyuel Zewdu Teferi, STMicroelectronics  
Suresh Venkatachalam, Synopsys, Inc.

## Past Contributors to v1.0:

David Antler, Intel Corporation  
Rajesh Bhaskar, Intel Corporation  
Ken Foust, Intel Corporation  
Amit Gil, Qualcomm Incorporated  
James Goel, Qualcomm Incorporated  
Arkadiusz Golec, Cadence Design Systems, Inc.  
Rob Gough, Intel Corporation  
Sharon Graif, Qualcomm Incorporated  
Wei Han, Lattice Semiconductor Corp.  
Michael Joehren, NXP Semiconductors  
Kenneth Ma, HiSilicon Technologies Co. Ltd.  
Mikko Muukki, HiSilicon Technologies Co. Ltd.  
Pratap Neelasheety, Synopsys, Inc.  
Ramiro Oliveira, Synopsys, Inc.  
Tomasz Pielaszkiewicz, Intel Corporation  
Radu Pitigoi-Aron, Qualcomm Incorporated  
Frank Seto, Samsung Electronics, Co.  
Jason Sher, Avery Design Systems, Inc.  
Satwant Singh, Lattice Semiconductor Corp.  
Vitor Soares, Synopsys, Inc.  
Przemyslaw Sroka, Cadence Design Systems, Inc.  
Dale Stolitzka, Samsung Electronics, Co.  
Nobu Suzuki, Intel Corporation  
Hiroo Takahashi, Sony Corporation  
Phani Uppalapati, Qualcomm Incorporated  
Suresh Venkatachalam, Synopsys, Inc.  
Girish Venkatraman, Intel Corporation  
Steve Watkins, WiSpry Inc.  
Rick Wietfeldt, Qualcomm Incorporated  
Weijiang Yang, Intel Corporation  
Shai Yosef, Qualcomm Incorporated