

Embedded Peripherals IP User Guide

Updated for Intel® Quartus® Prime Design Suite: **21.2**

Subscribe
Send Feedback

UG-01085 | 2021.06.28
Latest document on the web: PDF | HTML

Contents

1. Introduction..... 19
1.1. Tool Support..... 19
1.2. Device Support..... 20
1.3. Embedded Peripherals IP User Guide Archives..... 21
1.4. Document Revision History for Embedded Peripherals IP User Guide..... 22

2. Avalon-ST Multi-Channel Shared Memory FIFO Core..... 25
2.1. Core Overview..... 25
2.2. Performance and Resource Utilization..... 25
2.3. Functional Description..... 26
2.3.1. Interfaces..... 27
2.3.2. Operation..... 27
2.4. Parameters..... 28
2.5. Software Programming Model..... 29
2.5.1. HAL System Library Support..... 29
2.5.2. Register Map..... 29
2.6. Avalon-ST Multi-Channel Shared Memory FIFO Core Revision History..... 30

3. Avalon-ST Single-Clock and Dual-Clock FIFO Cores..... 31
3.1. Core Overview..... 31
3.2. Functional Description..... 31
3.2.1. Interfaces..... 32
3.2.2. Operating Modes..... 32
3.2.3. Fill Level..... 33
3.2.4. Thresholds..... 33
3.3. Parameters..... 34
3.4. Register Description..... 34
3.5. Avalon-ST Single-Clock and Dual-Clock FIFO Core Revision History..... 35

4. Avalon-ST Serial Peripheral Interface Core..... 37
4.1. Functional Description..... 37
4.1.1. Interfaces..... 37
4.1.2. Operation..... 38
4.1.3. Timing..... 39
4.1.4. Limitations..... 39
4.2. Configuration..... 39
4.3. Avalon-ST Serial Peripheral Interface Core Revision History..... 39

5. SPI Core..... 41
5.1. Core Overview..... 41
5.2. Functional Description..... 41
5.2.1. Example Configurations..... 43
5.2.2. Transmitter Logic..... 43
5.2.3. Receiver Logic..... 44
5.2.4. Host and Agent Modes..... 44
5.3. Configuration..... 47
5.3.1. Host/Agent Settings..... 47
5.3.2. Data Register Settings..... 48

5.3.3. Timing Settings..... 48
5.3.4. Synchronizer Stages.....49
5.4. Software Programming Model..... 49
5.4.1. Hardware Access Routines.....49
5.4.2. Software Files.....50
5.4.3. Register Map..... 51
5.5. Example Test Code.....54
5.6. SPI Core Revision History.....55

6. SPI Agent/JTAG to Avalon Host Bridge Cores..... 56
6.1. Core Overview.....56
6.2. Functional Description..... 57
6.3. Parameters..... 60
6.4. SPI Agent/JTAG to Avalon Host Bridge Cores Revision History..... 61

7. Intel eSPI Agent Core..... 62
7.1. Functional Description..... 63
7.1.1. Link Layer.....63
7.1.2. Transaction Layer.....65
7.1.3. Channel Specific Layer.....65
7.1.4. Port80 Implementation..... 68
7.1.5. VW message to Physical Port Implementation..... 68
7.1.6. Avalon Memory-Mapped Interface Interface Settings.....69
7.2. Resource Utilization.....70
7.3. IP Parameters..... 70
7.4. Interface Signals.....71
7.5. Registers..... 73
7.5.1. Avalon-MM Interface Accessible Registers.....73
7.5.2. eSPI Interface Accessible Registers..... 75
7.6. Peripheral Channel Avalon Interface Use Model..... 79
7.7. Intel eSPI Agent Core Revision History.....79

8. eSPI to LPC Bridge Core.....80
8.1. Unsupported LPC Features..... 80
8.2. IP Parameters..... 80
8.3. Supported IP Clock Frequency..... 81
8.4. Functional Description..... 82
8.4.1. FIFO Implementation.....82
8.4.2. Transaction Ordering Rule..... 83
8.4.3. eSPI Command to LPC Cycle Type Conversion.....83
8.4.4. SERIRQ Interrupt Event..... 84
8.5. Interface Signals.....86
8.6. Registers..... 88
8.6.1. Status Register.....88
8.6.2. Error Register.....89
8.7. eSPI to LPC Bridge Core Revision History..... 89

9. Ethernet MDIO Core..... 90
9.1. Core Overview.....90
9.2. Functional Description..... 90
9.2.1. MDIO Frame Format (Clause 45).....91
9.2.2. MDIO Clock Generation.....92

9.2.3. Interfaces....92
9.2.4. Operation....92
9.3. Parameter....93
9.4. Configuration Registers....93
9.5. Interface Signals....93
9.6. Ethernet MDIO Core Revision History....94
10. Intel FPGA 16550 Compatible UART Core....95
10.1. Core Overview....95
10.2. Feature Description....95
10.2.1. Unsupported Features....96
10.2.2. Interface....96
10.2.3. General Architecture....98
10.2.4. 16550 UART General Programming Flow Chart....98
10.2.5. Configuration Parameters....100
10.2.6. DMA Support....100
10.2.7. FPGA Resource Usage....101
10.2.8. Timing and Fmax....101
10.2.9. Avalon-MM Agent....102
10.2.10. Over-run/Under-run Conditions....103
10.2.11. Hardware Auto Flow-Control....104
10.2.12. Clock and Baud Rate Selection....105
10.3. Software Programming Model....105
10.3.1. Overview....105
10.3.2. Supported Features....105
10.3.3. Unsupported Features....106
10.3.4. Configuration....106
10.3.5. 16550 UART API....106
10.3.6. Driver Examples....110
10.4. Address Map and Register Descriptions....114
10.4.1. rbr_thr_dll....115
10.4.2. ier_dlh....116
10.4.3. iir....118
10.4.4. fcr....119
10.4.5. lcr....121
10.4.6. mcr....123
10.4.7. lsr....124
10.4.8. msr....126
10.4.9. scr....128
10.4.10. afr....129
10.4.11. tx_low....130
10.5. Intel FPGA 16550 Compatible UART Core Revision History....130
11. UART Core....132
11.1. Core Overview....132
11.2. Functional Description....132
11.2.1. Avalon-MM Agent Interface and Registers....132
11.2.2. RS-232 Interface....133
11.2.3. Transmitter Logic....133
11.2.4. Receiver Logic....133
11.2.5. Baud Rate Generation....134

11.3. Instantiating the Core....134
11.3.1. Configuration Settings....134
11.4. Software Programming Model....137
11.4.1. HAL System Library Support....137
11.4.2. Software Files....141
11.4.3. Register Map....141
11.4.4. Interrupt Behavior....146
11.5. UART Core Revision History....146

12. JTAG UART Core....148
12.1. Core Overview....148
12.2. Functional Description....148
12.2.1. Avalon Agent Interface and Registers....149
12.2.2. Read and Write FIFOs....149
12.2.3. JTAG Interface....149
12.2.4. Host-Target Connection....149
12.3. Configuration....150
12.3.1. Configuration Page....150
12.4. Software Programming Model....151
12.4.1. HAL System Library Support....151
12.4.2. Software Files....154
12.4.3. Accessing the JTAG UART Core via a Host PC....154
12.4.4. Register Map....154
12.4.5. Interrupt Behavior....156
12.5. JTAG UART Core Revision History....157

13. Intel FPGA Avalon Mailbox Core....158
13.1. Core Overview....158
13.2. Functional Description....158
13.2.1. Message Sending and Retrieval Process....159
13.2.2. Component Register Map....159
13.3. Interface....161
13.3.1. Component Interface....161
13.3.2. Component Parameterization....162
13.4. HAL Driver....163
13.4.1. Feature Description....163
13.5. Intel FPGA Avalon Mailbox Core Revision History....168

14. Intel FPGA Avalon Mutex Core....169
14.1. Core Overview....169
14.2. Functional Description....169
14.3. Configuration....170
14.4. Software Programming Model....170
14.4.1. Software Files....170
14.4.2. Hardware Access Routines....171
14.5. Mutex API....171
14.5.1. altera_avalon_mutex_is_mine()....171
14.5.2. altera_avalon_mutex_first_lock()....172
14.5.3. altera_avalon_mutex_lock()....172
14.5.4. altera_avalon_mutex_open()....172
14.5.5. altera_avalon_mutex_trylock()....173
14.5.6. altera_avalon_mutex_unlock()....173

14.6. Intel FPGA Avalon Mutex Core Revision History....173

15. Intel FPGA Avalon I²C (Host) Core....174

15.1. Core Overview....174
15.2. Feature Description....174
15.2.1. Supported Features....174
15.2.2. Unsupported Features....174
15.3. Configuration Parameters....174
15.4. Interface....175
15.5. Registers....176
15.5.1. Register Memory Map....176
15.5.2. Register Descriptions....177
15.6. Reset and Clock Requirements....181
15.7. Functional Description....181
15.7.1. Overview....181
15.7.2. Configuring TFT_CMD Register Examples....182
15.7.3. I²C Serial Interface Connection....185
15.7.4. Avalon-MM Agent Interface....186
15.7.5. Avalon-ST Interface....186
15.7.6. Programming Model....186
15.8. Intel FPGA Avalon I²C (Host) Core API....190
15.8.1. Optional Status Retrieval API....193
15.9. Intel FPGA Avalon I²C (Host) Core Revision History....195

16. Intel FPGA I²C Agent to Avalon-MM Host Bridge Core....196

16.1. Core Overview....196
16.2. Functional Description....196
16.2.1. Block Diagram....196
16.2.2. N-byte Addressing....196
16.2.3. N-byte Addressing with N-bit Address Stealing....197
16.2.4. Read Operation....198
16.2.5. Write Operation....199
16.2.6. Interacting with Multi-Hosts....201
16.3. Platform Designer Parameters....202
16.4. Signals....202
16.5. How to Translate the Bridge's I²C Data and I²C I/O Ports to an I²C Interface....204
16.6. Intel FPGA I²C Agent to Avalon-MM Host Bridge Core Revision History....205

17. Intel FPGA Avalon Compact Flash Core....206

17.1. Core Overview....206
17.2. Functional Description....206
17.3. Required Connections....207
17.4. Software Programming Model....208
17.4.1. HAL System Library Support....208
17.4.2. Software Files....208
17.4.3. Register Maps....209
17.5. Intel FPGA Avalon Compact Flash Core Revision History....210

18. EPCS/EPCQA Serial Flash Controller Core....211

18.1. Core Overview....211
18.2. Functional Description....212
18.2.1. Avalon-MM Agent Interface and Registers....214

18.3. **Configuration** 214
18.4. Interface Signals 215
18.5. Software Programming Model 215
18.5.1. HAL System Library Support 216
18.5.2. Software Files 216
18.6. Driver API 216
18.7. EPCS/EPCQA Serial Flash Controller Core Revision History 218

19. Intel FPGA Serial Flash Controller Core 219
19.1. Parameters 219
19.1.1. Configuration Device Types 219
19.1.2. I/O Mode 220
19.1.3. Chip Selects 220
19.1.4. Interface Signals 220
19.2. Registers 222
19.2.1. Register Memory Map 222
19.2.2. Register Descriptions 223
19.3. Nios II Tools Support 227
19.3.1. Booting Nios II from Flash 227
19.3.2. Nios II HAL Driver 229
19.4. Driver API 230
19.5. Intel FPGA Serial Flash Controller Core Revision History 232

20. Intel FPGA Serial Flash Controller II Core 233
20.1. Parameters 233
20.1.1. Configuration Device Types 233
20.1.2. I/O Mode 234
20.1.3. Chip Selects 234
20.1.4. Interface Signals 234
20.2. Registers 236
20.2.1. Register Memory Map 236
20.2.2. Register Descriptions 236
20.3. Nios II Tools Support 240
20.3.1. Booting Nios II from Flash 240
20.3.2. Nios II HAL Driver 243
20.4. Driver API 243
20.5. Intel FPGA Serial Flash Controller II Core Revision History 245

21. Intel FPGA Generic QUAD SPI Controller Core 246
21.1. Parameters 246
21.1.1. Configuration Device Types 247
21.1.2. I/O Mode 247
21.1.3. Chip Selects 247
21.1.4. Interface Signals 248
21.2. Registers 249
21.2.1. Register Memory Map 249
21.2.2. Register Descriptions 250
21.3. Nios II Tools Support 254
21.3.1. Nios II HAL Driver 254
21.4. Driver API 255
21.5. Intel FPGA Generic QUAD SPI Controller Core Revision History 257

22. Intel FPGA Generic QUAD SPI Controller II Core.....258
22.1. Parameters.....258
22.1.1. Disable Dedicated Active Serial Interface.....258
22.1.2. Enable SPI Pins Interface.....258
22.1.3. Enable Flash Simulation Model.....259
22.1.4. Configuration Device Types.....259
22.1.5. I/O Mode.....260
22.1.6. Chip Selects.....260
22.2. Interface Signals.....260
22.3. Registers.....261
22.3.1. Register Memory Map.....261
22.3.2. Register Descriptions.....262
22.4. Nios II Tools Support.....266
22.4.1. Nios II HAL Driver.....266
22.5. Driver API.....267
22.6. Intel FPGA Generic QUAD SPI Controller II Core Revision History.....269

23. Interval Timer Core.....270
23.1. Core Overview.....270
23.2. Functional Description.....270
23.2.1. Avalon-MM Agent Interface.....271
23.3. Configuration.....271
23.3.1. Timeout Period.....271
23.3.2. Counter Size.....272
23.3.3. Hardware Options.....272
23.3.4. Configuring the Timer as a Watchdog Timer.....273
23.4. Software Programming Model.....273
23.4.1. HAL System Library Support.....274
23.4.2. Software Files.....274
23.4.3. Register Map.....275
23.4.4. Interrupt Behavior.....277
23.5. Interval Time Core API.....277
23.6. Interval Timer Core Revision History.....278

24. Intel FPGA Avalon FIFO Memory Core.....279
24.1. Core Overview.....279
24.2. Functional Description.....279
24.2.1. Avalon-MM Write Agent to Avalon-MM Read Agent.....280
24.2.2. Avalon-ST Sink to Avalon-ST Source.....280
24.2.3. Avalon-MM Write Agent to Avalon-ST Source.....281
24.2.4. Avalon-ST Sink to Avalon-MM Read Agent.....283
24.2.5. Status Interface.....284
24.2.6. Clocking Modes.....284
24.3. Configuration.....284
24.3.1. FIFO Settings.....284
24.3.2. Interface Parameters.....284
24.3.3. Interface Signals.....285
24.4. Software Programming Model.....287
24.4.1. HAL System Library Support.....287
24.4.2. Software Files.....287
24.5. Programming with the Intel FPGA Avalon FIFO Memory.....288

24.5.1. Software Control 289
24.5.2. Software Example 291
24.6. Intel FPGA Avalon FIFO Memory API 292
24.6.1. altera_avalon_fifo_init() 292
24.6.2. altera_avalon_fifo_read_status() 292
24.6.3. altera_avalon_fifo_read_ienable() 293
24.6.4. altera_avalon_fifo_read_almostfull() 293
24.6.5. altera_avalon_fifo_read_almostempty() 293
24.6.6. altera_avalon_fifo_read_event() 293
24.6.7. altera_avalon_fifo_read_level() 294
24.6.8. altera_avalon_fifo_clear_event() 294
24.6.9. altera_avalon_fifo_write_ienable() 294
24.6.10. altera_avalon_fifo_write_almostfull() 295
24.6.11. altera_avalon_fifo_write_almostempty() 295
24.6.12. altera_avalon_write_fifo() 295
24.6.13. altera_avalon_write_other_info() 296
24.6.14. altera_avalon_fifo_read_fifo() 296
24.6.15. altera_avalon_fifo_read_other_info() 296
24.7. Intel FPGA Avalon FIFO Memory Core Revision History 297

25. On-Chip Memory (RAM and ROM) Core 298
25.1. Core Overview 298
25.2. Component-Level Design for On-Chip Memory 298
25.2.1. Memory Type 298
25.2.2. Size 299
25.2.3. Read Latency 299
25.2.4. ROM/RAM Memory Protection 300
25.2.5. ECC Parameter 300
25.2.6. Memory Initialization 300
25.3. Platform Designer System-Level Design for On-Chip Memory 300
25.4. Simulation for On-Chip Memory 300
25.5. Intel Quartus Prime Project-Level Design for On-Chip Memory 301
25.6. Board-Level Design for On-Chip Memory 301
25.7. Example Design with On-Chip Memory 301
25.8. On-Chip Memory (RAM and ROM) Core Revision History 301

26. Optrex 16207 LCD Controller Core 303
26.1. Core Overview 303
26.2. Functional Description 303
26.3. Software Programming Model 304
26.3.1. HAL System Library Support 304
26.3.2. Displaying Characters on the LCD 304
26.3.3. Software Files 305
26.3.4. Register Map 305
26.3.5. Interrupt Behavior 305
26.4. Optrex 16207 LCD Controller Core Revision History 306

27. PIO Core 307
27.1. Core Overview 307
27.2. Functional Description 307
27.2.1. Data Input and Output 308
27.2.2. Edge Capture 308

27.2.3. IRQ Generation 308
27.3. Example Configurations 309
27.3.1. Avalon-MM Interface 309
27.4. Configuration 309
27.4.1. Basic Settings 309
27.4.2. Input Options 310
27.4.3. Simulation 311
27.5. Software Programming Model 311
27.5.1. Software Files 311
27.5.2. Register Map 311
27.5.3. Interrupt Behavior 313
27.5.4. Software Files 313
27.6. PIO Core Revision History 314

28. PLL Cores 315
28.1. Core Overview 315
28.2. Functional Description 316
28.2.1. ALTPLL IP Core 316
28.2.2. Clock Outputs 316
28.2.3. PLL Status and Control Signals 317
28.2.4. System Reset Considerations 317
28.3. Instantiating the Avalon ALTPLL Core 317
28.4. Instantiating the PLL Core 317
28.5. Hardware Simulation Considerations 318
28.6. Register Definitions and Bit List 319
28.6.1. Status Register 319
28.6.2. Control Register 319
28.6.3. Phase Reconfig Control Register 320
28.7. PLL Cores Revision History 321

29. DMA Controller Core 322
29.1. Core Overview 322
29.2. Functional Description 322
29.2.1. Setting Up DMA Transactions 323
29.2.2. The Host Read and Write Ports 324
29.2.3. Addressing and Address Incrementing 324
29.3. Parameters 325
29.3.1. DMA Parameters (Basic) 325
29.3.2. Advanced Options 326
29.4. Software Programming Model 326
29.4.1. HAL System Library Support 326
29.4.2. Software Files 327
29.4.3. Register Map 328
29.4.4. Interrupt Behavior 331
29.5. DMA Controller Core Revision History 331

30. Modular Scatter-Gather DMA Core 332
30.1. Core Overview 332
30.2. Feature Description 332
30.3. mSGDMA Interfaces and Parameters 334
30.3.1. Interface 334
30.3.2. mSGDMA Parameter Editor 338

30.4. mSGDMA Descriptors....338
30.4.1. Read and Write Address Fields....339
30.4.2. Length Field....340
30.4.3. Sequence Number Field....340
30.4.4. Read and Write Burst Count Fields....340
30.4.5. Read and Write Stride Fields....340
30.4.6. Control Field....341
30.5. Register Map of mSGDMA....342
30.5.1. Status Register....343
30.5.2. Control Register....344
30.5.3. Write Fill Level Register....345
30.5.4. Read Fill Level Register....345
30.5.5. Response Fill Level Register....345
30.5.6. Write Sequence Number Register....345
30.5.7. Read Sequence Number Register....346
30.5.8. Component Configuration 1 Register....346
30.5.9. Component Configuration 2 Register....347
30.5.10. Component Type Register....348
30.5.11. Component Version Register....348
30.6. Programming Model....348
30.6.1. Stop DMA Operation....348
30.6.2. Stop Descriptor Operation....349
30.6.3. Recovery from Stopped on Error and Stopped on Early Termination....349
30.7. Modular Scatter-Gather DMA Prefetcher Core....350
30.7.1. Functional Description....350
30.8. Driver Implementation....363
30.8.1. alt_msgdma_standard_descriptor_async_transfer....363
30.8.2. alt_msgdma_extended_descriptor_async_transfer....364
30.8.3. alt_msgdma_descriptor_async_transfer....365
30.8.4. alt_msgdma_standard_descriptor_sync_transfer....366
30.8.5. alt_msgdma_extended_descriptor_sync_transfer....367
30.8.6. alt_msgdma_descriptor_sync_transfer....368
30.8.7. alt_msgdma_construct_standard_st_to_mm_descriptor....369
30.8.8. alt_msgdma_construct_standard_mm_to_st_descriptor....370
30.8.9. alt_msgdma_construct_standard_mm_to_mm_descriptor....371
30.8.10. alt_msgdma_construct_standard_descriptor....372
30.8.11. alt_msgdma_construct_extended_st_to_mm_descriptor....373
30.8.12. alt_msgdma_construct_extended_mm_to_st_descriptor....374
30.8.13. alt_msgdma_construct_extended_mm_to_mm_descriptor....375
30.8.14. alt_msgdma_construct_extended_descriptor....376
30.8.15. alt_msgdma_register_callback....377
30.8.16. alt_msgdma_open....378
30.8.17. alt_msgdma_write_standard_descriptor....379
30.8.18. alt_msgdma_write_extended_descriptor....380
30.8.19. alt_msgdma_init....381
30.8.20. alt_msgdma_irq....381
30.9. Example Code Using mSGDMA Core....381
30.10. Modular Scatter-Gather DMA Core Revision History....382

31. Scatter-Gather DMA Controller Core....383
31.1. Core Overview....383

31.1.1. Example Systems....383
31.1.2. Comparison of SG-DMA Controller Core and DMA Controller Core....385
31.2. Resource Usage and Performance....385
31.3. Functional Description....385
31.3.1. Functional Blocks and Configurations....386
31.3.2. DMA Descriptors....388
31.3.3. Error Conditions....390
31.4. Parameters....392
31.5. Simulation Considerations....392
31.6. Software Programming Model....392
31.6.1. HAL System Library Support....392
31.6.2. Software Files....393
31.6.3. Register Maps....393
31.6.4. DMA Descriptors....395
31.6.5. Timeouts....397
31.7. Programming with SG-DMA Controller....397
31.7.1. Data Structure....397
31.7.2. SG-DMA API....398
31.7.3. alt_avalon_sgdma_do_async_transfer()....398
31.7.4. alt_avalon_sgdma_do_sync_transfer()....399
31.7.5. alt_avalon_sgdma_construct_mem_to_mem_desc()....399
31.7.6. alt_avalon_sgdma_construct_stream_to_mem_desc()....400
31.7.7. alt_avalon_sgdma_construct_mem_to_stream_desc()....400
31.7.8. alt_avalon_sgdma_enable_desc_poll()....401
31.7.9. alt_avalon_sgdma_disable_desc_poll()....401
31.7.10. alt_avalon_sgdma_check_descriptor_status()....401
31.7.11. alt_avalon_sgdma_register_callback()....402
31.7.12. alt_avalon_sgdma_start()....402
31.7.13. alt_avalon_sgdma_stop()....402
31.7.14. alt_avalon_sgdma_open()....403
31.8. Scatter-Gather DMA Controller Core Revision History....403

32. SDRAM Controller Core....405
32.1. Core Overview....405
32.2. Functional Description....405
32.2.1. Avalon-MM Interface....406
32.2.2. Off-Chip SDRAM Interface....406
32.2.3. Board Layout and Pinout Considerations....408
32.2.4. Performance Considerations....408
32.3. Configuration....409
32.3.1. Memory Profile Page....409
32.3.2. Timing Page....411
32.4. Hardware Simulation Considerations....411
32.4.1. SDRAM Controller Simulation Model....412
32.4.2. SDRAM Memory Model....412
32.5. Example Configurations....412
32.6. Software Programming Model....414
32.7. Clock, PLL and Timing Considerations....414
32.7.1. Factors Affecting SDRAM Timing....414
32.7.2. Symptoms of an Untuned PLL....415
32.7.3. Estimating the Valid Signal Window....415

32.7.4. Example Calculation....416
32.8. SDRAM Controller Core Revision History....419

33. Tri-State SDRAM Core....420
33.1. Core Overview....420
33.2. Feature Description....420
33.2.1. Block Diagram....421
33.3. Configuration Parameter....421
33.3.1. Memory Profile Page....421
33.3.2. Timing Page....421
33.4. Interface....422
33.5. Reset and Clock Requirements....425
33.6. Architecture....425
33.6.1. Avalon-MM Agent Interface and CSR....425
33.6.2. Block Level Usage Model....426
33.7. Intel SDRAM Tri-State Controller Core Revision History....426

34. Video Sync Generator and Pixel Converter Cores....427
34.1. Core Overview....427
34.2. Video Sync Generator....427
34.2.1. Functional Description....427
34.2.2. Parameters....428
34.2.3. Signals....429
34.2.4. Timing Diagrams....429
34.3. Pixel Converter....430
34.3.1. Functional Description....430
34.3.2. Parameters....431
34.3.3. Signals....431
34.4. Hardware Simulation Considerations....431
34.5. Video Sync Generator and Pixel Converter Cores Revision History....431

35. Intel FPGA Interrupt Latency Counter Core....433
35.1. Core Overview....433
35.2. Feature Description....433
35.2.1. Avalon-MM Compliant CSR Registers....434
35.2.2. 32-bit Counter....436
35.2.3. Interrupt Detector....436
35.3. Component Interface....436
35.4. Component Parameterization....437
35.5. Software Access....437
35.5.1. Routine for Level Sensitive Interrupts....437
35.5.2. Routine for Edge/Pulse Sensitive Interrupts....437
35.6. Implementation Details....438
35.6.1. Interrupt Latency Counter Architecture....438
35.7. IP Caveats....439
35.8. Intel FPGA Interrupt Latency Counter Core Revision History....439

36. Performance Counter Unit Core....440
36.1. Core Overview....440
36.2. Functional Description....440
36.2.1. Section Counters....440
36.2.2. Global Counter....441

36.2.3. Register Map....441
36.2.4. System Reset....442
36.3. Configuration....442
36.3.1. Define Counters....442
36.3.2. Multiple Clock Domain Considerations....442
36.4. Hardware Simulation Considerations....442
36.5. Software Programming Model....442
36.5.1. Software Files....442
36.5.2. Using the Performance Counter....442
36.5.3. Interrupt Behavior....444
36.6. Performance Counter API....445
36.6.1. PERF_RESET()....445
36.6.2. PERF_START_MEASURING()....445
36.6.3. PERF_STOP_MEASURING()....445
36.6.4. PERF_BEGIN()....446
36.6.5. PERF_END()....446
36.6.6. perf_print_formatted_report()....446
36.6.7. perf_get_total_time()....447
36.6.8. perf_get_section_time()....447
36.6.9. perf_get_num_starts()....447
36.6.10. alt_get_cpu_freq()....448
36.7. Performance Counter Core Revision History....448

37. Vectored Interrupt Controller Core....449
37.1. Core Overview....449
37.2. Functional Description....451
37.2.1. External Interfaces....451
37.2.2. Functional Blocks....452
37.2.3. Daisy Chaining VIC Cores....454
37.2.4. Latency Information....454
37.3. Register Maps....455
37.4. Parameters....459
37.5. Intel FPGA HAL Software Programming Model....460
37.5.1. Software Files....460
37.5.2. Macros....460
37.5.3. Data Structure....461
37.5.4. VIC API....461
37.5.5. Run-time Initialization....463
37.5.6. Board Support Package....463
37.6. Implementing the VIC in Platform Designer....469
37.6.1. Adding VIC Hardware....469
37.6.2. Software for VIC....474
37.7. Example Designs....476
37.7.1. Example Description....476
37.7.2. Example Usage....478
37.7.3. Software Description....478
37.7.4. Positioning the ISR in Vector Table....480
37.7.5. Latency Measurement with the Performance Counter....481
37.8. Advanced Topics....482
37.8.1. Real Time Latency Concerns....482
37.8.2. Software Interrupt....486

37.9. Vectored Interrupt Controller Core Revision History....487
38. Avalon-ST Data Pattern Generator and Checker Cores....488
38.1. Core Overview....488
38.2. Data Pattern Generator....488
38.2.1. Functional Description....488
38.2.2. Configuration....490
38.3. Data Pattern Checker....490
38.3.1. Functional Description....490
38.3.2. Configuration....492
38.4. Hardware Simulation Considerations....493
38.5. Software Programming Model....493
38.5.1. Register Maps....493
38.6. Avalon-ST Data Pattern Generator and Checker Cores Revision History....497
39. Avalon-ST Test Pattern Generator and Checker Cores....498
39.1. Core Overview....498
39.2. Resource Utilization and Performance....498
39.3. Test Pattern Generator....499
39.3.1. Functional Description....499
39.3.2. Configuration....500
39.4. Test Pattern Checker....501
39.4.1. Functional Description....501
39.4.2. Configuration....502
39.5. Hardware Simulation Considerations....503
39.6. Software Programming Model....503
39.6.1. HAL System Library Support....503
39.6.2. Software Files....503
39.6.3. Register Maps....503
39.7. Test Pattern Generator API....507
39.7.1. data_source_reset()....507
39.7.2. data_source_init()....507
39.7.3. data_source_get_id()....508
39.7.4. data_source_get_supports_packets()....508
39.7.5. data_source_get_num_channels()....508
39.7.6. data_source_get_symbols_per_cycle()....508
39.7.7. data_source_set_enable()....509
39.7.8. data_source_get_enable()....509
39.7.9. data_source_set_throttle()....509
39.7.10. data_source_get_throttle()....509
39.7.11. data_source_is_busy()....510
39.7.12. data_source_fill_level()....510
39.7.13. data_source_send_data()....510
39.8. Test Pattern Checker API....511
39.8.1. data_sink_reset()....511
39.8.2. data_sink_init()....511
39.8.3. data_sink_get_id()....511
39.8.4. data_sink_get_supports_packets()....511
39.8.5. data_sink_get_num_channels()....512
39.8.6. data_sink_get_symbols_per_cycle()....512
39.8.7. data_sink_set enable()....512

39.8.8. data_sink_get_enable()....512
39.8.9. data_sink_set_throttle()....513
39.8.10. data_sink_get_throttle()....513
39.8.11. data_sink_get_packet_count()....513
39.8.12. data_sink_get_symbol_count()....513
39.8.13. data_sink_get_error_count()....514
39.8.14. data_sink_get_exception()....514
39.8.15. data_sink_exception_is_exception()....514
39.8.16. data_sink_exception_has_data_error()....514
39.8.17. data_sink_exception_has_missing_sop()....515
39.8.18. data_sink_exception_has_missing_eop()....515
39.8.19. data_sink_exception_signalled_error()....515
39.8.20. data_sink_exception_channel()....515
39.9. Avalon-ST Test Pattern Generator and Checker Cores Revision History....516

40. System ID Peripheral Core....517
40.1. Core Overview....517
40.2. Functional Description....517
40.3. Configuration....518
40.4. Software Programming Model....518
40.4.1. alt_avalon_sysid_test()....519
40.5. System ID Core Revision History....519

41. Avalon Packets to Transactions Converter Core....520
41.1. Core Overview....520
41.2. Functional Description....520
41.2.1. Interfaces....520
41.2.2. Operation....521
41.3. Avalon Packets to Transactions Converter Core Revision History....522

42. Avalon-ST Multiplexer and Demultiplexer Cores....524
42.1. Core Overview....524
42.1.1. Resource Usage and Performance....524
42.2. Multiplexer....525
42.2.1. Functional Description....525
42.2.2. Parameters....526
42.3. Demultiplexer....527
42.3.1. Functional Description....527
42.3.2. Parameters....528
42.4. Hardware Simulation Considerations....529
42.5. Software Programming Model....529
42.6. Avalon-ST Multiplexer and Demultiplexer Cores Revision History....529

43. Avalon-ST Bytes to Packets and Packets to Bytes Converter Cores....531
43.1. Core Overview....531
43.2. Functional Description....531
43.2.1. Interfaces....532
43.2.2. Operation—Avalon-ST Bytes to Packets Converter Core....532
43.2.3. Operation—Avalon-ST Packets to Bytes Converter Core....533
43.3. Avalon-ST Bytes to Packets and Packets to Bytes Converter Cores Revision History... 533

44. Avalon-ST Delay Core.....535
44.1. Core Overview.....535
44.2. Functional Description.....535
44.2.1. Reset.....535
44.2.2. Interfaces.....536
44.3. Parameters.....536
44.4. Avalon-ST Delay Core Revision History.....537
45. Avalon-ST Round Robin Scheduler Core.....538
45.1. Core Overview.....538
45.2. Performance and Resource Utilization.....538
45.3. Functional Description.....539
45.3.1. Interfaces.....539
45.3.2. Operations.....540
45.4. Parameters.....541
45.5. Avalon-ST Round Robin Scheduler Core Revision History.....541
46. Avalon-ST Splitter Core.....542
46.1. Core Overview.....542
46.2. Functional Description.....542
46.2.1. Backpressure.....542
46.2.2. Interfaces.....543
46.3. Parameters.....543
46.4. Avalon-ST Splitter Core Revision History.....544
47. Avalon-MM DDR Memory Half Rate Bridge Core.....546
47.1. Core Overview.....546
47.2. Resource Usage and Performance.....547
47.3. Functional Description.....547
47.4. Instantiating the Core in Platform Designer.....548
47.5. Example System.....549
47.6. Avalon-MM DDR Memory Half Rate Bridge Core Revision History.....550
48. Intel FPGA GMII to RGMII Converter Core.....551
48.1. Core Overview.....551
48.2. Feature Description.....551
48.2.1. Supported Features.....551
48.2.2. Unsupported Features.....551
48.3. Parameters.....551
48.3.1. IP Configuration Parameter.....551
48.4. Intel FPGA GMII to RGMII Converter Core Interface.....552
48.5. Functional Description.....554
48.5.1. Architecture.....555
48.6. Intel FPGA HPS EMAC Interface Splitter Core.....556
48.6.1. Parameter.....556
48.7. Intel FPGA GMII to RGMII Converter Core Revision History.....562
49. Intel FPGA MII to RMII Converter Core.....563
49.1. Supported Features.....563
49.2. Interface Signals.....564
49.3. Parameters.....565
49.4. Functional Description.....565

49.4.1. Clocking Scheme....565
49.4.2. Transmit Interface....566
49.4.3. Receive Interface....567
49.5. Intel FPGA MII to RMII Converter Core Revision History....567

50. Intel FPGA HPS GMII to TSE 1000BASE-X/SGMII PCS Bridge Core....568
50.1. Core Overview....568
50.2. Feature Description....568
50.2.1. Supported Features....569
50.3. Core Architecture....569
50.3.1. Data Path....569
50.3.2. Clock Scheme....570
50.3.3. MAC Speed....570
50.3.4. Transmit Elastic Buffer....570
50.3.5. Avalon-MM Agent Interface....571
50.3.6. Programming Model....571
50.4. Configuration Parameters....572
50.5. Interface....572
50.6. Registers....574
50.6.1. Register Memory Map....574
50.6.2. Register Description....574
50.7. Intel FPGA HPS GMII to TSE 1000BASE-X/SGMII PCS Bridge Core Revision History....575

51. Intel FPGA HPS EMAC to Multi-rate PHY GMII Adapter Core....576
51.1. Supported Features....576
51.2. Functional Description....576
51.3. Interface Signals....578
51.4. Register Memory Map and Description....579
51.5. Intel FPGA HPS EMAC to Multi-rate PHY GMII Adapter Core Revision History....580

52. Intel FPGA MSI to GIC Generator Core....581
52.1. Core Overview....581
52.2. Background....581
52.3. Feature Description....581
52.3.1. Interrupt Servicing Process....583
52.3.2. Registers of Component....583
52.3.3. Unsupported Feature....584
52.4. Intel FPGA MSI to GIC Generator Core Revision History....585

1. Introduction

This user guide describes the IP cores provided by Intel® Quartus® Prime design software.

The IP cores are optimized for Intel FPGA devices and can be easily implemented to reduce design and test time. You can use the IP parameter editor from Platform Designer to add the IP cores to your system, configure the cores, and specify their connectivity.

Before using Platform Designer, review the Intel Quartus Prime software Release Notes for known issues and limitations. To submit general feedback or technical support, click **Feedback** on the Intel Quartus Prime software **Help** menu and also on all Intel FPGA technical documentation.

Related Information

- Intel Quartus Prime Pro Edition Handbook
- Intel Quartus Prime Standard Edition Handbook

1.1. Tool Support

Platform Designer is a system-level integration tool which is included as part of the Intel Quartus Prime software. Platform Designer saves significant time and effort in the FPGA design process by automatically generating interconnect logic to connect intellectual property (IP) functions and subsystems. You can implement a design using the IP cores from the Platform Designer component library.

All the IP cores described in this user guide are supported by both Intel Quartus Prime Pro Edition and Intel Quartus Prime Standard Edition except for the following cores which are only supported by Intel Quartus Prime Standard Edition.

- SDRAM Controller Core
- Tri-State SDRAM Core
- Compact Flash Core
- EPCS Serial Flash Controller Core
- 16207 LCD Controller Core
- Scatter-Gather DMA Controller Core
- Video Sync Generator and Pixel Converter Cores
- Avalon®-ST Test Pattern Generator and Checker Cores
- Avalon-MM DDR Memory Half Rate Bridge Core
- Modular ADC Core
- Modular Dual ADC Core

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Note: Intel Quartus Prime Pro Edition only supports Intel Agilex™, Intel Stratix® 10, Intel Arria® 10, and Intel Cyclone® 10 GX device families.

Related Information

- Intel Quartus Prime Pro Edition Software and Device Support Release Notes
- Intel Quartus Prime Standard Edition Software and Device Support Release Notes

1.2. Device Support

The following IP cores are only supported in Intel Quartus Prime Standard Edition:

- EPCS Serial Flash Controller Core
- Scatter-Gather DMA Controller Core
- Video Sync Generator and Pixel Converter Cores
- Avalon Streaming Interface Test Pattern Generator and Checker Cores
- Modular ADC Core (only for Intel MAX® 10 devices)
- Modular Dual ADC Core (only for Intel MAX 10 devices)

Note: Intel Quartus Prime Standard Edition supports older FPGA devices: Stratix V and below, Arria V and below, Cyclone V and below. For more information, refer to *Intel Quartus Prime Standard Edition Software and Device Support Release Notes*.

The following IPs are part of a product obsolescence and support discontinuation schedule:

- Intel FPGA Avalon Compact Flash Core
- Optrex 16207 LCD Controller Core
- SDRAM Controller Core
- Tri-State SDRAM Core
- Avalon-MM DDR Memory Half Rate Bridge Core

Added support for Intel eASIC™ N5X devices in:

- Avalon-ST Serial Peripheral Interface Core
- SPI Agent to Avalon Host Bridge Core
- Intel FPGA 16550 Compatible UART Core
- Intel FPGA Avalon I^2C (Host) Core
- Intel FPGA I^2C Agent to Avalon-MM Host Bridge Core
- PIO Core
- Intel FPGA HPS EMAC Interface Splitter Core
- Intel FPGA HPS GMII to TSE 1000BASE-X/SGMII PCS Bridge Core

Other IP cores described in this user guide support all Intel FPGA device families unless specified in their respective chapter.

Different device families support different I/O standards, which may affect the ability of the core to interface to certain components. For details about supported I/O types, refer to the device handbook for the target device family.

Related Information

- Intel Quartus Prime Pro Edition Software and Device Support Release Notes
- Intel Quartus Prime Standard Edition Software and Device Support Release Notes

1.3. Embedded Peripherals IP User Guide Archives

If an IP core version is not listed, the user guide for the previous IP core version applies.

IP Core Version	User Guide
21.1	Embedded Peripherals IP User Guide
20.3	Embedded Peripherals IP User Guide
20.2	Embedded Peripherals IP User Guide
19.4	Embedded Peripherals IP User Guide
19.2	Embedded Peripherals IP User Guide
19.1	Embedded Peripherals IP User Guide
18.1	Embedded Peripherals IP User Guide
18.0	Embedded Peripherals IP User Guide

1.4. Document Revision History for Embedded Peripherals IP User Guide

This section covers the revision history of the entire volume. For details regarding changes to a specific chapter refer to their respective chapter revision history.

Document Version	Intel Quartus Prime Version	Changes
2021.06.28	21.2	• Replaced non-inclusive terms "master" and "slave" to inclusive terms "host" and "agent" respectively for the following interface descriptions: — Avalon — SPI — I^2C • Added a new section: *Embedded Peripherals IP User Guide Archives*.
2021.03.29	21.1	• Added support for Intel eASIC N5X devices in: — *Avalon-ST Serial Peripheral Interface Core* — *SPI Agent to Avalon Host Bridge Core* — *Intel FPGA 16550 Compatible UART Core* — *Intel FPGA Avalon I^2C (Host) Core* — *Intel FPGA I^2C Agent to Avalon-MM Host Bridge Core* — *PIO Core* — *Intel FPGA HPS EMAC Interface Splitter Core* — *Intel FPGA HPS GMII to TSE 1000BASE-X/SGMII PCS Bridge Core* • Updated the following chapters: — *SPI Core* — *Intel FPGA Serial Flash Controller Core* — *Intel FPGA Serial Flash Controller II Core* — *Intel FPGA Generic QUAD SPI Controller Core* — *Intel FPGA Generic QUAD SPI Controller II Core* • The following IPs are part of a product obsolescence and support discontinuation schedule: — *Intel FPGA Avalon Compact Flash Core* — *Optrex 16207 LCD Controller Core* — *SDRAM Controller Core* — *Tri-State SDRAM Core* — *Avalon-MM DDR Memory Half Rate Bridge Core*
2020.12.13	20.3	Updated the following chapter: • *Vectored Interrupt Controller Core*
2020.09.21	20.2	Updated the following chapter: • *eSPI to LPC Bridge Core* • *Modular Scatter-Gather DMA Core* • *Avalon-ST Data Pattern Generator and Checker Cores*
2020.07.22	20.2	Updated the following chapter: • *Intel eSPI Agent Core* • *eSPI to LPC Bridge Core* • *Intel FPGA MII to RMII Converter Core*
2020.01.22	19.4	Updated the following chapter: • *UART Core*
2019.12.16	19.4	Updated the following chapter:

continued...

Document Version	Intel Quartus Prime Version	Changes
		• *Avalon-ST Single-Clock and Dual-Clock FIFO Core* • *SPI Core* • *SDRAM Controller Core* • *Intel FPGA Serial Flash Controller II Core* • *Intel FPGA Generic QUAD SPI Controller II Core* Added a new chapter: *Intel FPGA MII to RMII Converter Core*.
2019.08.16	19.2	Updated the following chapter: • *Intel FPGA GMII to RGMII Converter Core*
2019.07.16	19.1	Updated the following chapters: • *Avalon-ST Serial Peripheral Interface Core* • *Intel FPGA 16550 Compatible UART Core* • *Intel FPGA Avalon FIFO Memory Core*
2019.04.01	19.1	• Added a new chapter: *Intel FPGA HPS EMAC to Multi-rate PHY GMII Adapter Core*. • Updated the following chapters: — *SPI Agent/JTAG to Avalon Host Bridge Cores* — *Intel FPGA Serial Flash Controller II Core* — *Intel FPGA Generic QUAD SPI Controller Core* — *Intel FPGA Generic QUAD SPI Controller II Core* — *Modular Scatter-Gather DMA Core*
2018.09.24	18.1	• Added a new chapter: *eSPI to LPC Bridge Core*. • Updated the following chapters: — *SPI Core* — *eSPI Core* — *UART Core* — *Intel FPGA Avalon I²C (Host) Core* — *EPCS/EPCQA Serial Flash Controller Core* — *Intel FPGA Serial Flash Controller Core* — *Intel FPGA Serial Flash Controller II Core* — *Intel FPGA Generic QUAD SPI Controller Core* — *Intel FPGA Generic QUAD SPI Controller II Core* — *Interval Timer Core* — *On-Chip FIFO Memory Core* — *Modular Scatter-Gather DMA Core*
2018.05.07	18.0	• Added new chapter *eSPI Core*. • Implemented editorial enhancement for all chapters.

Date	Version	Changes
November 2017	2017.11.06	• Removed the *Supported Devices* topic from various cores. Refer to *Device Support*.
December 2016	2016.12.19	Maintenance release.
October 2016	2016.10.28	New chapters: • Altera Avalon I²C (Host) Core Updated: • 16550 UART Core • Altera I²C Agent to Avalon-MM Host Bridge Core
June 2016	2016.06.17	New chapters: • Avalon-MM DDR Memory Half Rate Bridge Core

continued...

Date	Version	Changes
		Updated chapters: • UART Core • SPI Core • Altera Interrupt Latency Counter Core • Altera I²C Agent to Avalon-MM Host Bridge Core
May 2016	2016.05.03	New chapters: • Altera I²C Agent to Avalon-MM Host Bridge Core Updated chapters: • Vectored Interrupt Controller Core
December 2015	2015.12.16	Removed chapters: • PCI Lite Core • Avalon-ST JTAG Interface Core Updated chapters: • 16550 UART Core • PIO Core • Altera Modular Scatter-Gather DMA Core
November 2015	2015.11.06	Removed chapters: • Mailbox Core-Replaced with Intel FPGA Avalon Mailbox Core on page 158 Updated chapters: • 16550 UART Core • Avalon-ST Bytes to Packets and Packets to Bytes Converter Cores • Altera Modular Scatter-Gather DMA Core • Vectored Interrupt Controller Core • Altera GMII to RGMII Adapter Core • Altera Avalon Mailbox (simple) Core
June 2015	2015.06.12	New chapters: • Altera Quad SPI Controller Core • Altera Serial Flash Controller Core • Altera Avalon Mailbox Core • Altera GMII to RGMII Adapter Core Updated chapters: • 16550 UART Core • Performance Counter Core • DMA Controller Core • PIO Core • Interval Timer Core The following chapters have been reinserted: • Avalon-ST Single-Clock and Dual-Clock FIFO Cores • Avalon Streaming Channel Multiplexer and Demultiplexer Cores • Avalon-ST Round Robin Scheduler Core • Avalon-ST Delay Core • Avalon-ST Splitter Core • Avalon Streaming Test Pattern Generator and Checker Cores • Avalon Streaming Data Pattern Generator and Checker Cores The following chapters have been removed: • Common Flash Interface Controller Core • Cyclone III Remote Update Controller Core (No longer available starting from V14.0)

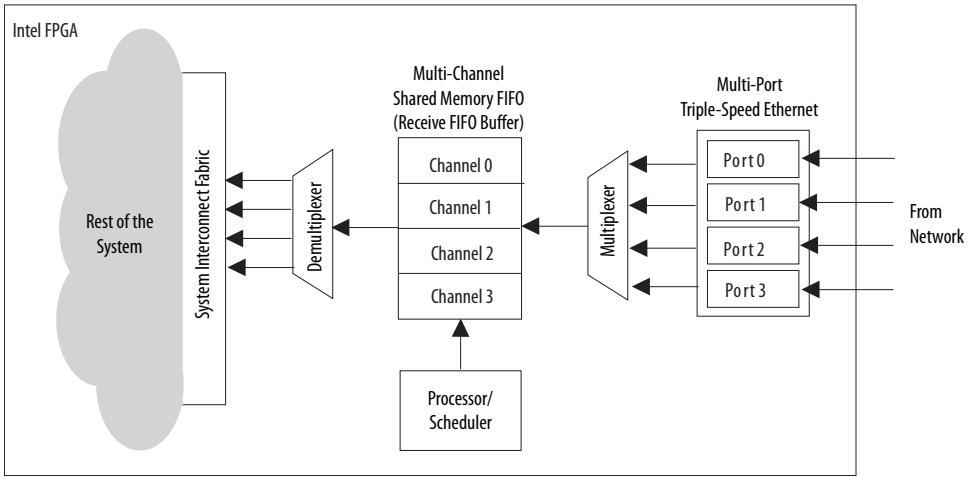
2. Avalon-ST Multi-Channel Shared Memory FIFO Core

2.1. Core Overview

The Avalon Streaming (Avalon-ST) Multi-Channel Shared Memory FIFO core is a FIFO buffer with Avalon-ST data interfaces. The core, which supports up to 16 channels, is a contiguous memory space with dedicated segments of memory allocated for each channel. Data is delivered to the output interface in the same order it was received on the input interface for a given channel.

The example below shows an example of how the core is used in a system. In this example, the core is used to buffer data going into and coming from a four-port Triple Speed Ethernet IP Core. A processor, if used, can request data for a particular channel to be delivered to the Triple Speed Ethernet IP Core.

Figure 1. Multi-Channel Shared Memory FIFO in a System—An Example



2.2. Performance and Resource Utilization

This section lists the resource utilization and performance data for various Intel FPGA device families. The estimates are obtained by compiling the core using the Intel Quartus Prime software.

The table below shows the resource utilization and performance data for a Stratix II GX device (EP2SGX130GF1508I4).

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Table 1. Memory Utilization and Performance Data for Stratix II GX Devices

Channels	ALUTs	Logic Registers	Memory Blocks			f_{MAX} (MHz)
			M512	M4K	M-RAM	
4	559	382	0	0	1	> 125
12	1617	1028	0	0	6	> 125

The table below shows the resource utilization and performance data for a Stratix III device (EP3SL340F1760C3). The performance of the IP Core in Stratix IV devices is similar to Stratix III devices.

Table 2. Memory Utilization and Performance Data for Stratix III Devices

Channels	ALUTs	Logic Registers	Memory Blocks			f_{MAX} (MHz)
			M9K	M144K	MLAB	
4	557	345	37	0	0	> 125
12	1741	1028	0	24	0	> 125

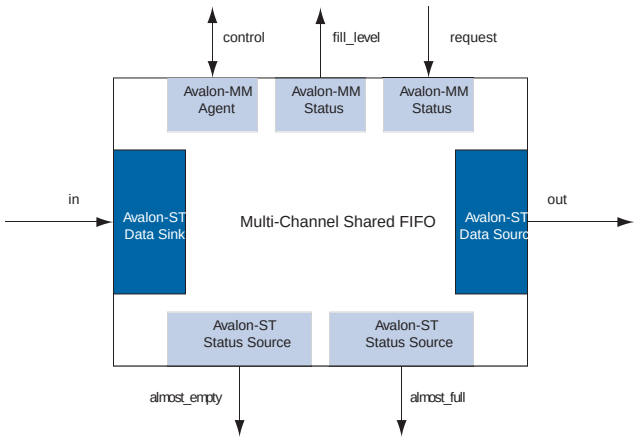
The table below shows the resource utilization and performance data for a Cyclone III device (EP3C120F780I7).

Table 3. Memory Utilization and Performance Data for Cyclone III Devices

Channels	Total Logic Elements	Total Registers	Memory M9K	f_{MAX} (MHz)
4	711	346	37	> 125
12	2284	1029	412	> 125

2.3. Functional Description

Figure 2. Avalon-ST Multi-Channel Shared Memory FIFO Core



2.3.1. Interfaces

This section describes the core's interfaces.

Avalon-ST Interfaces

The core includes Avalon-ST interfaces for transferring data and almost-full status.

Table 4. Properties of Avalon-ST Interfaces

Feature	Property	
	Data Interfaces	Status Interfaces
Backpressure	Ready latency = 0.	Not supported.
Data Width	Configurable.	Data width = 2 bits. Symbols per beat = 1.
Channel	Supported, up to 16 channels.	Supported, up to 16 channels.
Error	Configurable.	Not used.
Packet	Supported.	Not supported.

Avalon-MM Interfaces

The core can have up to three Avalon-MM interfaces:

- **Avalon-MM control interface**—Allows host peripherals to set and access almost-full and almost-empty thresholds. The same set of thresholds is used by all channels. See **Control Interface Register Map** figure for the description of the threshold registers.
- **Avalon-MM fill-level interface**—Allows host peripherals to retrieve the fill level of the FIFO buffer for a given channel. The fill level represents the amount of data in the FIFO buffer at any given time. The read latency on this interface is one. See the **Fill-level Interface Register Map** table for the description of the fill-level registers.
- **Avalon-MM request interface**—Allows host peripherals to request data for a given channel. This interface is implemented only when the **Use Request** parameter is turned on. The `request_address` signal contains the channel number. Only one word of data is returned for each request.

  For more information about Avalon interfaces, refer to the Avalon Interface Specifications.

2.3.2. Operation

The Avalon-ST Multi-Channel Shared FIFO core allocates dedicated memory segments within the core for each channel, and is implemented such that the memory segments occupy a single memory block. The parameter **FIFO depth** determines the depth of each memory segment.

The core receives data on its `in` interface (Avalon-ST sink) and stores the data in the allocated memory segments. If a packet contains any error (`in_error` signal is asserted), the core drops the packet.

When the core receives a request on its `request` interface (Avalon-MM agent), it forwards the requested data to its `out` interface (Avalon-ST source) only when it has received a full packet on its `in` interface. If the core has not received a full packet or has no data for the requested channel, it deasserts the `valid` signal on its `out` interface to indicate that data is not available for the channel. The output latency is three and only one word of data can be requested at a time.

When the Avalon-MM request interface is not in use, the `request_write` signal is kept asserted and the `request_address` signal is set to 0. Hence, if you configure the core to support more than one channel, you must also ensure that the **Use request** parameter is turned on. Otherwise, only channel 0 is accessible.

You can configure almost-full thresholds to manage FIFO overflow. The current threshold status for each channel is available from the core's Avalon-ST status interfaces in a round-robin fashion. For example, if the threshold status for channel 0 is available on the interface in clock cycle n, the threshold status for channel 1 is available in clock cycle n+1 and so forth.

2.4. Parameters

Table 5. Configurable Parameters

Parameter	Legal Values	Description
Number of channels	1, 2, 4, 8, and 16	The total number of channels supported on the Avalon-ST data interfaces.
Symbols per beat	1–32	The number of symbols transferred in a beat on the Avalon-ST data interfaces
Bits per symbol	1–32	The symbol width in bits on the Avalon-ST data interfaces.
Error width	0–32	The width of the `error` signal on the Avalon-ST data interfaces.
FIFO depth	2–2^{32}	The depth of each memory segment allocated for a channel. The value must be a multiple of 2.
Use packets	0 or 1	Setting this parameter to 1 enables packet support on the Avalon-ST data interfaces.
Use fill level	0 or 1	Setting this parameter to 1 enables the Avalon-MM status interface.
Number of almost-full thresholds	0 to 2	The number of almost-full thresholds to enable. Setting this parameter to 1 enables **Use almost-full threshold 1**. Setting it to 2 enables both **Use almost-full threshold 1** and **Use almost-full threshold 2**.
Number of almost-empty thresholds	0 to 2	The number of almost-empty thresholds to enable. Setting this parameter to 1 enables **Use almost-empty threshold 1**. Setting it to 2 enables both **Use almost-empty threshold 1** and **Use almost-empty threshold 2**.
Section available threshold	0 to $2^{\text{Address Width}}$	Specify the amount of data to be delivered to the output interface. This parameter applies only when packet support is disabled.
Packet buffer mode	0 or 1	Setting this parameter to 1 causes the core to deliver only full packets to the output interface. This parameter applies only when **Use packets** is set to 1.
Drop on error	0 or 1	Setting this parameter to 1 causes the core to drop packets at the Avalon-ST data sink interface if the `error` signal on that interface is asserted. Otherwise, the core accepts the packet and sends it out on the Avalon-ST data source interface with the same error. This parameter applies only when packet buffer mode is enabled.
Address width	1–32	The width of the FIFO address. This parameter is determined by the parameter **FIFO depth**; **FIFO depth** = $2^{\text{Address Width}}$.

continued...

Parameter	Legal Values	Description
Use request	—	Turn on this parameter to implement the Avalon-MM request interface. If the core is configured to support more than one channel and the request interface is disabled, only channel 0 is accessible.
Use almost-full threshold 1	—	Turn on these parameters to implement the optional Avalon-ST almost-full and almost-empty interfaces and their corresponding registers. See **Control Interface Register Map** for the description of the threshold registers.
Use almost-full threshold 2	—	
Use almost-empty threshold 1	—	
Use almost-empty threshold 2	—	
Use almost-full threshold 1	0 or 1	This threshold indicates that the FIFO is almost full. It is enabled when the parameter **Number of almost-full threshold** is set to 1 or 2.
Use almost-full threshold 2	0 or 1	This threshold is an initial indication that the FIFO is getting full. It is enabled when the parameter **Number of almost-full threshold** is set to 2.
Use almost-empty threshold 1	0 or 1	This threshold indicates that the FIFO is almost empty. It is enabled when the parameter **Number of almost-empty threshold** is set to 1 or 2.
Use almost-empty threshold 2	0 or 1	This threshold is an initial indication that the FIFO is getting empty. It is enabled when the parameter **Number of almost-empty threshold** is set to 2.

2.5. Software Programming Model

The following sections describe the software programming model for the Avalon-ST Multi-Channel Shared FIFO core.

2.5.1. HAL System Library Support

The Intel-provided driver implements a HAL device driver that integrates into the HAL system library for Nios® II systems. HAL users should access the Avalon-ST Multi-Channel Shared FIFO core via the familiar HAL API and the ANSI C standard library.

2.5.2. Register Map

You can configure the thresholds and retrieve the fill-level for each channel via the Avalon-MM control and fill-level interfaces respectively. Subsequent sections describe the registers accessible via each interface.

Control Register Interface

Table 6. Control Interface Register Map

Byte Offset	Name	Access	Reset Value	Description
0	ALMOST_FULL_THRESHOLD	RW	0	Primary almost-full threshold. The bit `Almost_full_data[0]` on the Avalon-ST almost-full status interface is set to 1 when the FIFO level is equal to or greater than this threshold.
4	ALMOST_EMPTY_THRESHOLD	RW	0	Primary almost-empty threshold. The bit `Almost_empty_data[0]` on the Avalon-ST almost-empty status interface is set to 1 when the FIFO level is equal to or less than this threshold.

continued...

Byte Offset	Name	Access	Reset Value	Description
8	ALMOST_FULL2_THRESHOLD	RW	0	Secondary almost-full threshold. The bit `Almost_full_data[1]` on the Avalon-ST almost-full status interface is set to 1 when the FIFO level is equal to or greater than this threshold.
12	ALMOST_EMPTY2_THRESHOLD	RW	0	Secondary almost-empty threshold. The bit `Almost_empty_data[1]` on the Avalon-ST almost-empty status interface is set to 1 when the FIFO level is equal to or less than this threshold.
Base + 8	`Almost_Empty_Threshold`	RW		The value of the primary almost-empty threshold. The bit `Almost_empty_data[0]` on the Avalon-ST almost-empty status interface is set to 1 when the FIFO level is greater than or equal to this threshold.
Base + 12	`Almost_Empty2_Threshold`	RW		The value of the secondary almost-empty threshold. The bit `Almost_empty_data[1]` Avalon-ST almost-empty status interface is set to 1 when the FIFO level is greater than or equal to this threshold.

Fill-Level Register Interface

The table below shows the register map for the fill-level interface.

Table 7. Fill-level Interface Register Map

Byte Offset	Name	Access	Reset Value	Description
0	fill_level_0	RO	0	Fill level for each channel. Each register is defined for each channel. For example, if the core is configured to support four channel, four fill-level registers are defined.
4	fill_level_1	RO	0	
8	fill_level_2	RO	0	
(n*4)	fill_level_n	RO	0	

2.6. Avalon-ST Multi-Channel Shared Memory FIFO Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
December 2010	v10.1.0	Removed the “Device Support”, “Instantiating the Core in SOPC Builder”, and “Referenced Documents” sections.
July 2010	v10.0.0	Added the description of almost-empty thresholds and fill-level registers. Revised the Operation section.
November 2009	v9.1.0	No change from previous release.
March 2009	v9.0.0	No change from previous release.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	v8.0.0	Initial release.

3. Avalon-ST Single-Clock and Dual-Clock FIFO Cores

3.1. Core Overview

The Avalon Streaming (Avalon-ST) Single-Clock and Avalon-ST Dual-Clock FIFO cores are FIFO buffers which operate with a common clock and independent clocks for input and output ports respectively. The FIFO cores are configurable, Platform Designer-ready, and integrate easily into any Platform Designer-generated systems.

3.2. Functional Description

The following two figures show block diagrams of the Avalon-ST Single-Clock FIFO and Avalon-ST Dual-Clock FIFO cores.

Figure 3. Avalon-ST Single Clock FIFO Core

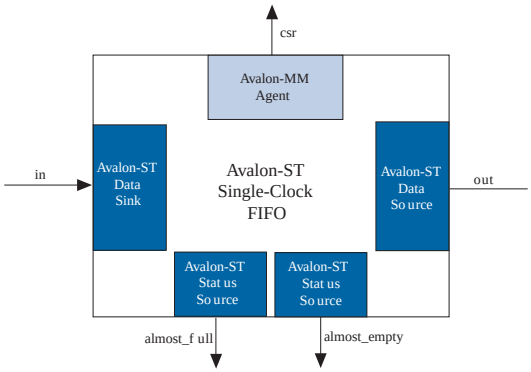
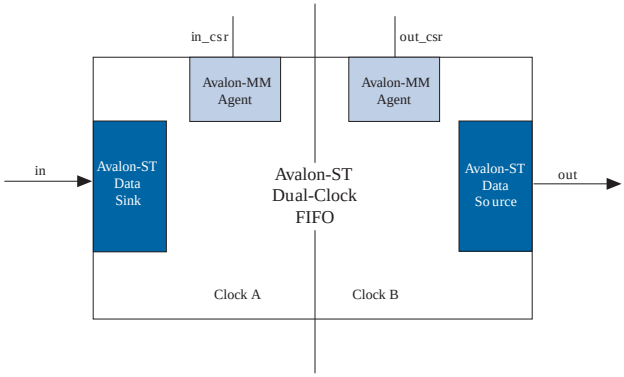


Figure 4. Avalon-ST Dual Clock FIFO Core



Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

3.2.1. Interfaces

This section describes the interfaces implemented in the FIFO cores.

Avalon-ST Data Interface

Each FIFO core has an Avalon-ST data sink and source interfaces. The data sink and source interfaces in the dual-clock FIFO core are driven by different clocks.

Table 8. Properties of Avalon-ST Interfaces

Feature	Property
Backpressure	Ready latency = 0.
Data Width	Configurable.
Channel	Supported, up to 255 channels.
Error	Configurable.
Packet	Configurable.

Avalon-MM Control and Status Register Interface

You can configure the single-clock FIFO core to include an optional Avalon-MM interface, and the dual-clock FIFO core to include an Avalon-MM interface in each clock domain. The Avalon-MM interface provides access to 32-bit registers, which allows you to retrieve the FIFO buffer fill level and configure the almost-empty and almost-full thresholds. In the single-clock FIFO core, you can also configure the packet and error handling modes.

Avalon-ST Status Interface

The single-clock FIFO core has two optional Avalon-ST status source interfaces from which you can obtain the FIFO buffer almost-full and almost empty statuses.

Related Information

Avalon Interface Specifications
For more information about Avalon interfaces.

3.2.2. Operating Modes

The following lists the FIFO operating modes:

- Default mode—The core accepts incoming data on the `in` interface (Avalon Streaming Interface data sink) and forwards it to the `out` interface (Avalon Streaming Interface data source). The core asserts the `valid` signal on the Avalon Streaming Interface source interface to indicate that data is available at the interface.
- Store and forward mode—This mode only applies to the single-clock FIFO core. The core asserts the `valid` signal on the `out` interface only when a full packet of data is available at the interface.

  In this mode, you can also enable the drop-on-error feature by setting the `drop_on_error` register to 1. When this feature is enabled, the core drops all packets received with the `in_error` signal asserted.
- Cut-through mode— This mode only applies to the single-clock FIFO core. The core asserts the `valid` signal on the `out` interface to indicate that data is available for consumption when the number of entries specified in the `cut_through_threshold` register are available in the FIFO buffer.

  To use the store and forward or cut-through mode, turn on the **Use store and forward** parameter to include the `csr` interface (Avalon Memory-Mapped Interface agent). Set the `cut_through_threshold` register to 0 to enable the store and forward mode; set the register to any value greater than 0 to enable the cut-through mode. The non-zero value specifies the minimum number of FIFO entries that must be available before the data is ready for consumption. Setting the register to 1 provides you with the default mode.

3.2.3. Fill Level

You can obtain the fill level of the FIFO buffer via the optional Avalon Memory-Mapped Interface control and status interface. Turn on the **Use fill level** parameter (**Use sink fill level** and **Use source fill level** in the dual-clock FIFO core) and read the `fill_level` register.

The dual-clock FIFO core has two fill levels, one in each clock domain. Due to the latency of the clock crossing logic, the fill levels reported in the input and output clock domains may be different at any given instance. In both cases, the fill level is pessimistic for the clock domain; the fill level is reported high in the input clock domain and low in the output clock domain.

The dual-clock FIFO has an output pipeline stage to improve f_{MAX}. This output stage is accounted for when calculating the output fill level, but not when calculating the input fill level. Hence, the best measure of the amount of data in the FIFO is given by the fill level in the output clock domain, while the fill level in the input clock domain represents the amount of space available in the FIFO (Available space = **FIFO depth** – input fill level).

3.2.4. Thresholds

You can use almost-full and almost-empty thresholds as a mechanism to prevent FIFO overflow and underflow. This feature is only available in the single-clock FIFO core.

To use the thresholds, turn on the **Use fill level**, **Use almost-full status**, and **Use almost-empty status** parameters. You can access the `almost_full_threshold` and `almost_full_threshold` registers via the csr interface and set the registers to an optimal value for your application.

You can obtain the almost-full and almost-empty statuses from `almost_full` and `almost_empty` interfaces (Avalon Streaming Interface status source). The core asserts the `almost_full` signal when the fill level is equal to or higher than the almost-full threshold. Likewise, the core asserts the `almost_empty` signal when the fill level is equal to or lower than the almost-empty threshold.

3.3. Parameters

Table 9. Configurable Parameters

Parameter	Legal Values	Description
Bits per symbol	1–32	These parameters determine the width of the FIFO. FIFO width = **Bits per symbol** * **Symbols per beat**, where: Bits per symbol is the number of bits in a symbol, and Symbols per beat is the number of symbols transferred in a beat.
Symbols per beat	1–32	
Error width	0–32	The width of the `error` signal.
FIFO depth	1–32	The FIFO depth. An output pipeline stage is added to the FIFO to increase performance, which increases the FIFO depth by one.
Use packets	—	Turn on this parameter to enable packet support on the Avalon-ST data interfaces.
Channel width	1–32	The width of the `channel` signal.
Avalon-ST Single Clock FIFO Only		
Use fill level	—	Turn on this parameter to include the Avalon-MM control and status register interface.
Avalon-ST Dual Clock FIFO Only		
Use sink fill level	—	Turn on this parameter to include the Avalon-MM control and status register interface in the input clock domain.
Use source fill level	—	Turn on this parameter to include the Avalon-MM control and status register interface in the output clock domain.
Write pointer synchronizer length	2–8	The length of the write pointer synchronizer chain. Setting this parameter to a higher value leads to better metastability while increasing the latency of the core.
Read pointer synchronizer length	2–8	The length of the read pointer synchronizer chain. Setting this parameter to a higher value leads to better metastability.
Use Max Channel	—	Turn on this parameter to specify the maximum channel number.
Max Channel	1–255	Maximum channel number.

For more information on metastability in Intel FPGA devices, refer to AN 42: Metastability in Intel FPGA devices.

For more information on metastability analysis and synchronization register chains, refer to the Area and Timing Optimization chapter in volume 2 of the *Intel Quartus Prime Handbook*.

3.4. Register Description

The `csr` interface in the Avalon-ST Single Clock FIFO core provides access to registers. The table below describes the registers.

Table 10. Register Description for Avalon-ST Single-Clock FIFO

32-Bit Word Offset	Name	Access	Reset	Description
0	`fill_level`	R	0	24-bit FIFO fill level. Bits 24 to 31 are unused.
1	Reserved	—	—	Reserved for future use.
2	almost_full_threshold	RW	**FIFO depth**–1	Set this register to a value that indicates the FIFO buffer is getting full.
3	almost_empty_threshold	RW	0	Set this register to a value that indicates the FIFO buffer is getting empty.
4	cut_through_threshold	RW	0	0—Enables store and forward mode. >0—Enables cut-through mode and specifies the minimum of entries in the FIFO buffer before the `valid` signal on the Avalon-ST source interface is asserted. Once the FIFO core starts sending the data to the downstream component, it continues to do so until the end of the packet. This register applies only when the **Use store and forward** parameter is turned on.
5	drop_on_error	RW	0	0—Disables drop-on error. 1—Enables drop-on error. This register applies only when the **Use packet** and **Use store and forward** parameters are turned on.

The `in_csr` and `out_csr` interfaces in the Avalon-ST Dual Clock FIFO core reports the FIFO fill level. The table below describes the fill level.

Table 11. Register Description for Avalon-ST Dual-Clock FIFO

32-Bit Word Offset	Name	Access	Reset Value	Description
0	`fill_level`	R	0	24-bit FIFO fill level. Bits 24 to 31 are unused.

3.5. Avalon-ST Single-Clock and Dual-Clock FIFO Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2019.12.16	19.4	Corrected the *Register Description for Avalon-ST Dual-Clock FIFO*.
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
December 2010	v10.1.0	Removed the "Device Support", "Instantiating the Core in SOPC Builder", and "Referenced Documents" sections.
July 2010	v10.0.0	Added description of the new features of the single-clock FIFO: store and forward mode, cut-through mode, and drop on error. Added parameters and registers.

continued...

Date	Version	Changes
November 2009	v9.1.0	No change from previous release.
March 2009	v9.0.0	Added description of new parameters, **Write pointer synchronizer length** and **Read pointer synchronizer length**.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	v8.0.0	Initial release.

4. Avalon-ST Serial Peripheral Interface Core

The Avalon Streaming (Avalon-ST) Serial Peripheral Interface (SPI) core is an SPI agent that allows data transfers between Platform Designer systems and off-chip SPI devices via Avalon-ST interfaces. Data is serially transferred on the SPI, and sent to and received from the Avalon-ST interface in bytes.

The SPI Agent to Avalon Host Bridge is an example of how this core is used.

For more information on the bridge, refer to *SPI Agent/JTAG to Avalon Host Bridge Cores*.

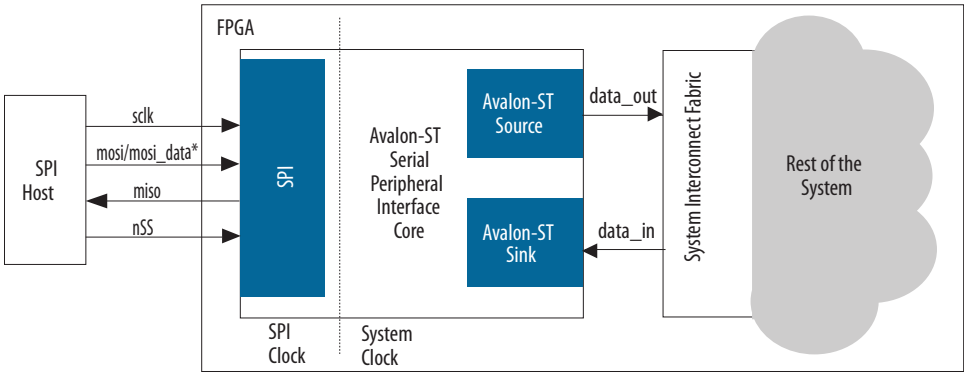
Note: This IP core supports Intel eASIC N5X devices.

Related Information

SPI Agent/JTAG to Avalon Host Bridge Cores on page 56

4.1. Functional Description

Figure 5. System with an Avalon-ST SPI Core



Note: For the Intel eASIC N5X device, the signal `miso_data` (output port) is generated instead of the `miso` (inout port) signal. You must instantiate eASIC IO pad modules to create tristate logic to the `miso_data` to create bidirectional signal.

4.1.1. Interfaces

The serial peripheral interface is full-duplex and does not support backpressure. It supports SPI clock phase bit, CPHA = 1, and SPI clock polarity bit, CPOL = 0.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Table 12. Properties of Avalon-ST Interfaces

Feature	Property
Backpressure	Not supported.
Data Width	Data width = 8 bits; Bits per symbol = 8.
Channel	Not supported.
Error	Not used.
Packet	Not supported.

For more information about Avalon-ST interfaces, refer to the Avalon Interface Specifications.

4.1.2. Operation

The Avalon-ST SPI core waits for the nSS signal to be asserted low, signifying that the SPI host is initiating a transaction. The core then starts shifting in bits from the input signal `mosi`. The core packs the bits received on the SPI to bytes and checks for the following special characters:

- `0x4a`—Idle character. The core drops the idle character.
- `0x4d`—Escape character. The core drops the escape character, and XORs the following byte with `0x20`.

  For each valid byte of data received, the core asserts the `valid` signal on its Avalon-ST source interface and presents the byte on the interface for a clock cycle.

  At the same time, the core shifts data out from the Avalon-ST sink to the output signal `miso` beginning with from the most significant bit. If there is no data to shift out, the core shifts out idle characters (`0x4a`). If the data is a special character, the core inserts an escape character (`0x4d`) and XORs the data with `0x20`.

  The data shifts into and out of the core in the direction of MSB first.

Figure 6. SPI Transfer Protocol

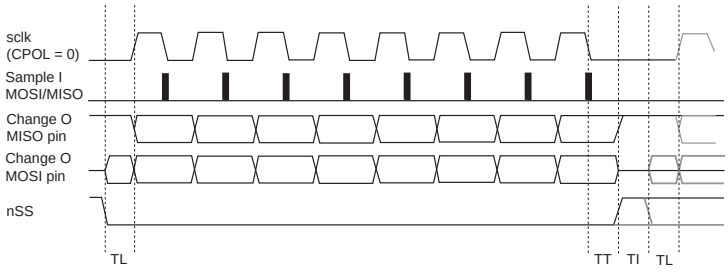


Table 13. SPI Transfer Protocol Specification

Symbol	Description	Minimum	Maximum	Unit
TL	The worst recovery time of `sclk` with respect with nSS.	½ SPI clock	-	Clock cycle
TT	The worst hold time for `MOSI` and `MISO` data.	½ SPI clock	-	
TI	The minimum width of a reset pulse required by Intel FPGA families.	1 SPI clock	-	

4.1.3. Timing

The core requires a lead time (TL) between asserting the nSS signal and the SPI clock, and a lag time (TT) between the last edge of the SPI clock and deasserting the nSS signal. The nSS signal must be deasserted for a minimum idling time (TI) of one SPI clock between byte transfers. A Timing Analyzer SDC file (.**sdc**) is provided to remove false timing paths. The frequency of the SPI host's clock must be equal to or lower than the frequency of the core's clock.

4.1.4. Limitations

Daisy-chain configuration, where the output line `miso` of an instance of the core is connected to the input line `mosi` of another instance, is not supported.

4.2. Configuration

The parameter **Number of synchronizer stages: Depth** allows you to specify the length of synchronization register chains. These register chains are used when a metastable event is likely to occur and the length specified determines the meantime before failure. The register chain length, however, affects the latency of the core.

For more information on metastability in Intel FPGA devices, refer to AN 42: Metastability in Intel FPGA devices.

For more information on metastability analysis and synchronization register chains, refer to the Area and Timing Optimization chapter in volume 2 of the *Intel Quartus Prime Handbook*.

4.3. Avalon-ST Serial Peripheral Interface Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2021.03.29	21.1	Added support for Intel eASIC N5X devices.
2019.07.16	19.1	Add the SPI Transfer Protocol Specification in *Operation* section.
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
July 2014	2014.07.24	Removed mention of SOPC Builder, updated to Platform Designer
December 2010	v10.1.0	Removed the "Device Support", "Instantiating the Core in SOPC Builder", and "Referenced Documents" sections.
July 2010	v10.0.0	No change from previous release.
November 2009	v9.1.0	Added a description to specify the shift direction.
March 2009	v9.0.0	Added description of a new parameter, **Number of synchronizer stages: Depth**.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	v8.0.0	Initial release.

5. SPI Core

5.1. Core Overview

SPI is an industry-standard serial protocol commonly used in embedded systems to connect microprocessors to a variety of off-chip sensor, conversion, memory, and control devices. The SPI core with Avalon interface implements the SPI protocol and provides an Avalon Memory-Mapped (Avalon-MM) interface on the back end.

The SPI core can implement either the host or agent protocol. When configured as a host, the core can control up to 32 independent SPI agents. The width of the receive and transmit registers are configurable between 1 and 32 bits. Longer transfer lengths can be supported with software routines. The core provides an interrupt output that can flag an interrupt whenever a transfer completes.

5.2. Functional Description

The SPI core communicates using two data lines, a control line, and a synchronization clock:

- Host Out Agent In (`mosi`)—Output data from the host to the inputs of the agents
- Host In Agent Out (`miso`)—Output data from a agent to the input of the host
- Serial Clock (`sclk`)—Clock driven by the host to agents, used to synchronize the data bits
- Agent Select (`ss_n`)— Select signal (active low) driven by the host to individual agents, used to select the target agent

The SPI core has the following user-visible features:

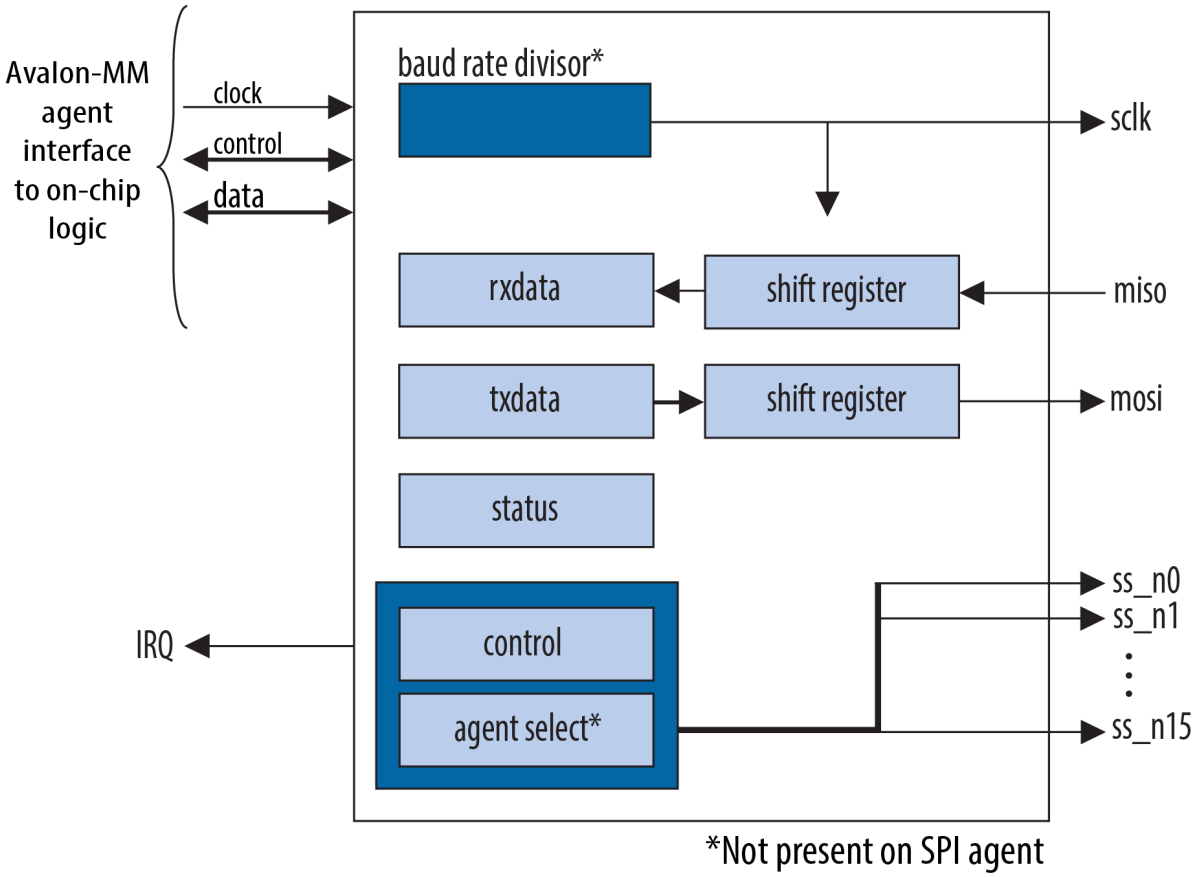
- A memory-mapped register space comprised of five registers: `rxdata`, `txdata`, `status`, `control`, and `slaveselect`
- Four SPI interface ports: `sclk`, `ss_n`, `mosi`, and `miso`

The registers provide an interface to the SPI core and are visible via the Avalon-MM agent port. The `sclk`, `ss_n`, `mosi`, and `miso` ports provide the hardware interface to other SPI devices. The behavior of `sclk`, `ss_n`, `mosi`, and `miso` depends on whether the SPI core is configured as a host or agent.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Figure 7. SPI Core Block Diagram (Host Mode)



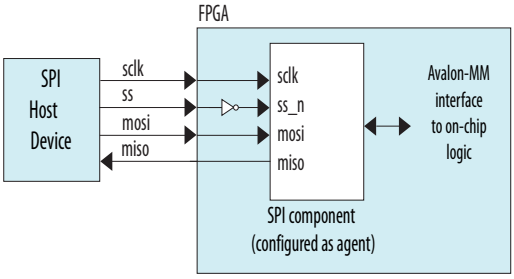
The SPI core logic is synchronous to the clock input provided by the Avalon-MM interface. When configured as a host, the core divides the Avalon-MM clock to generate the SCLK output. When configured as a agent, the core's receive logic is synchronized to SCLK input.

For more details, refer to the "Interval Timer Core" chapter.

5.2.1. Example Configurations

The core block diagram and the SPI core configured as a agent diagram show two possible configurations. In Figure 8 on page 43 the core provides a agent interface to an off-chip SPI host.

Figure 8. SPI Core Configured as a Agent



In the SPI core block diagram, the SPI core provides a host interface driving multiple off-chip agent devices. Each agent device in Figure 8 on page 43 must tristate its `miso` output whenever its select signal is not asserted.

The `ss_n` signal is active-low. However, any signal can be inverted inside the FPGA, allowing the agent-select signals to be either active high or active low.

5.2.2. Transmitter Logic

The core transmitter logic consists of a transmit holding register (`txdata`) and transmit shift register, each n bits wide. The register width n is specified at system generation time, and can be any integer value from 8 to 32. After a host peripheral writes a value to the `txdata` register, the value is copied to the shift register and then transmitted when the next operation starts.

The shift register and the `txdata` register provide double buffering during data transmission. A new value can be written into the `txdata` register while the previous data is being shifted out of the shift register. The transmitter logic automatically transfers the `txdata` register to the shift register whenever a serial shift operation is not currently in process.

In host mode, the transmit shift register directly feeds the `mosi` output. In agent mode, the transmit shift register directly feeds the `miso` output. Data shifts out LSB first or MSB first, depending on the configuration of the SPI core.

5.2.3. Receiver Logic

The core receive logic consists of a receive holding register (`rxdata`) and receive shift register, each n bits wide. The register width n is specified at system generation time, and can be any integer value from 8 to 32. A host peripheral reads received data from the `rxdata` register after the shift register has captured a full n-bit value of data.

The shift register and the `rxdata` register provide double buffering while receiving data. The `rxdata` register can hold a previously received data value while subsequent new data is shifting into the shift register. The receiver logic automatically transfers the shift register content to the `rxdata` register when a serial shift operation completes.

In host mode, the shift register is fed directly by the `miso` input. In agent mode, the shift register is fed directly by the `mosi` input. The receiver logic expects input data to arrive LSB first or MSB first, depending on the configuration of the SPI core.

5.2.4. Host and Agent Modes

At system generation time, the designer configures the SPI core in either host mode or agent mode. The mode cannot be switched at runtime.

5.2.4.1. Host Mode Operation

In host mode, the SPI ports behave as shown in the table below.

Table 14. Host Mode Port Configurations

Name	Direction	Description
`mosi`	output	Data output to agent(s)
`miso`	input	Data input from agent(s)
`sclk`	output	Synchronization clock to all agents
`ss_nM`	output	Agent select signal to agent M, where M is a number between 0 and 31.

In host mode, an intelligent host (for example, a microprocessor) configures the SPI core using the `control` and `slaveselect` registers, and then writes data to the `txdata` buffer to initiate a transaction. A host peripheral can monitor the status of the transaction by reading the `status` register. A host peripheral can enable interrupts to notify the host whenever new data is received (for example, a transfer has completed), or whenever the transmit buffer is ready for new data.

The SPI protocol is full duplex, so for every transaction both sends and receives data at the same time. The host transmits a new data bit on the `mosi` output and the agent drives a new data bit on the `miso` input for each active edge of `sclk`. The SPI core divides the Avalon-MM system clock using a clock divider to generate the `sclk` signal.

When the SPI core is configured to interface with multiple agents, the core has one `ss_n` signal for each agent. During a transfer, the host asserts `ss_n` to each agent specified in the `slaveselect` register. Note that there can be no more than one

agent transmitting data during any particular transfer, or else there will be a contention on the `miso` input. The number of agent devices is specified at system generation time.

5.2.4.2. Agent Mode Operation

In agent mode, the SPI ports behave as shown in the table below.

Table 15. Agent Mode Port Configurations

Name	Direction	Description
`mosi`	input	Data input from the host
`miso`	output	Data output to the host
`sclk`	input	Synchronization clock
`ss_n`	input	Select signal

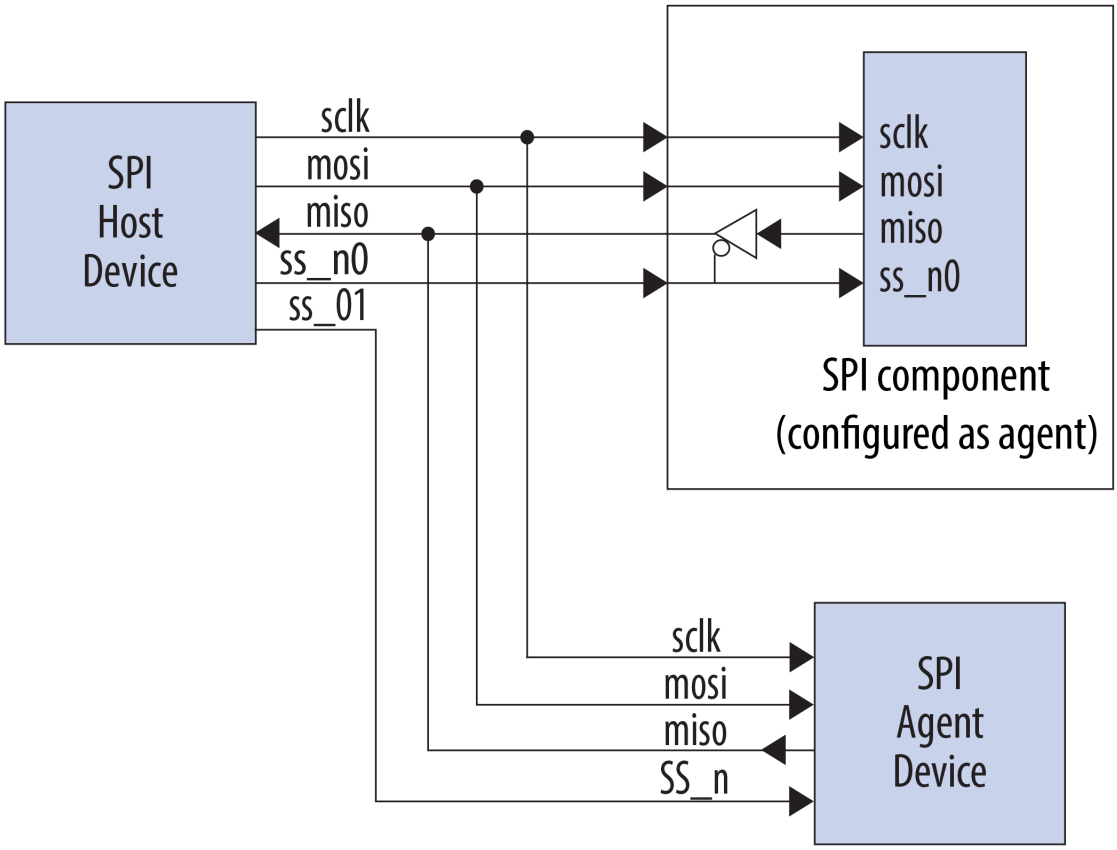
In agent mode, the SPI core simply waits for the host to initiate transactions. Before a transaction begins, the agent logic continuously polls the `ss_n` input. When the host asserts `ss_n`, the agent logic immediately begins sending the transmit shift register contents to the `miso` output. The agent logic also captures data on the `mosi` input, and fills the receive shift register simultaneously. After a word is received by the agent, the host must de-assert the `ss_n` signal and reasserts the signal again when the next word is ready to be sent.

An intelligent host such as a microprocessor writes data to the `txdata` registers, so that it is transmitted the next time the host initiates an operation. A host peripheral reads received data from the `rxdata` register. A host peripheral can enable interrupts to notify the host whenever new data is received, or whenever the transmit buffer is ready for new data.

5.2.4.3. Multi-Agent Environments

When `ss_n` is not asserted, typical SPI cores set their `miso` output pins to high impedance. The provided SPI agent core drives an undefined high or low value on its `miso` output when not selected. Special consideration is necessary to avoid signal contention on the `miso` output, if the SPI core in agent mode is connected to an off-chip SPI host device with multiple agents. In this case, the `ss_n` input should be used to control a tristate buffer on the `miso` signal.

Figure 9. SPI Core in a Multi-Agent Environment


SPI
Host
Device
sclk
mosi
miso
ss_n0
ss_01
sclk
mosi
miso
ss_n0
SPI component
(configured as agent)
sclk
mosi
miso
SS_n
SPI
Agent
Device

5.3. Configuration

The following sections describe the available configuration options.

5.3.1. Host/Agent Settings

The designer can select either host mode or agent mode to determine the role of the SPI core. When host mode is selected, the following options are available: **Number of select (SS_n) signals**, **SPI clock rate**, and **Specify delay**.

5.3.1.1. Number of Select (SS_n) Signals

This setting specifies the number of agents the SPI host connects to. The range is 1 to 32. The SPI host core presents a unique `ss_n` signal for each agent.

5.3.1.2. SPI Clock (sclk) Rate

This setting determines the rate of the `sclk` signal that synchronizes data between host and agents. The target clock rate can be specified in units of Hz, kHz or MHz. The SPI host core uses the Avalon-MM system clock and a clock divisor to generate `sclk`.

The actual frequency of `sclk` may not exactly match the desired target clock rate. The achievable clock values are:

<Avalon-MM system clock frequency> / [2, 4, 6, 8, ...]

The actual frequency achieved will not be greater than the specified target value.

Note: When the SPI core is turned on with a synchronizer, you must maintain the requirement against the sampling edge of SCLK, which is equal to (`<synchronizer depth> IP clock + 1 IP clock`). For example, if you select synchronizer depth of 2, this results in a sampling edge requirement of 3 clock cycles.

5.3.1.3. Specify Delay

Turning on this option causes the SPI host to add a time delay between asserting the `ss_n` signal and shifting the first bit of data. This delay is required by certain SPI agent devices. If the delay option is on, you must also specify the delay time in units of ns, µs or ms. An example is shown in below.

Figure 10. Time Delay Between Asserting ss_n and Toggling sclk



The delay generation logic uses a granularity of half the period of `sclk`. The actual delay achieved is the desired target delay rounded up to the nearest multiple of half the `sclk` period, as shown in the follow two equations.

Table 16.

p = 1/2 x (period of `sclk`)

Table 17.

Actual delay = ceiling x (desired delay/ p)

5.3.2. Data Register Settings

The data register settings affect the size and behavior of the data registers in the SPI core. There are two data register settings:

- **Width**—This setting specifies the width of `rxdata`, `txdata`, and the receive and transmit shift registers. The range is from 1 to 32.
- **Shift direction**—This setting determines the direction that data shifts (MSB first or LSB first) into and out of the shift registers.

5.3.3. Timing Settings

The timing settings affect the timing relationship between the `ss_n`, `sclk`, `mosi` and `miso` signals. In this discussion the `mosi` and `miso` signals are referred to generically as data. There are two timing settings:

- **Clock polarity**—This setting can be 0 or 1. When clock polarity is set to 0, the idle state for `sclk` is low. When clock polarity is set to 1, the idle state for `sclk` is high.
- **Clock phase**—This setting can be 0 or 1. When clock phase is 0, data is latched on the leading edge of `sclk`, and data changes on trailing edge. When clock phase is 1, data is latched on the trailing edge of `sclk`, and data changes on the leading edge.

  The following four clock polarity figures demonstrate the behavior of signals in all possible cases of clock polarity and clock phase.

Figure 11. Clock Polarity = 0, Clock Phase = 0

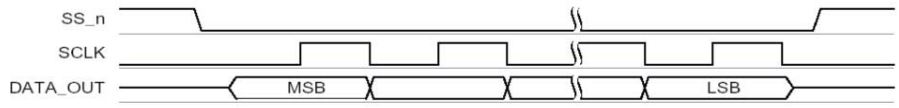


Figure 12. Clock Polarity = 0, Clock Phase = 1

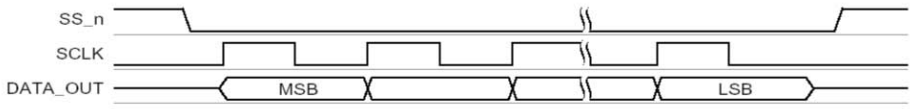


Figure 13. Clock Polarity = 1, Clock Phase = 0

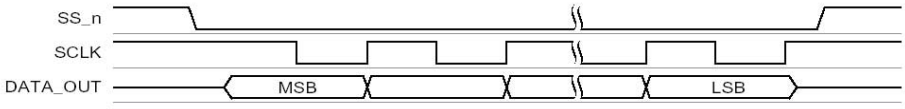
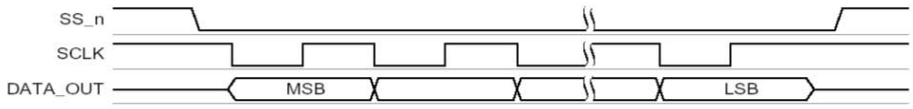


Figure 14. Clock Polarity = 1, Clock Phase = 1



5.3.4. Synchronizer Stages

This setting enables a synchronizer on the SPI interface input. When agent mode is used, you must turn on the synchronizer.

5.4. Software Programming Model

The following sections describe the software programming model for the SPI core, including the register map and software constructs used to access the hardware. For Nios II processor users, Intel provides the HAL system library header file that defines the SPI core registers. The SPI core does not match the generic device model categories supported by the HAL, so it cannot be accessed via the HAL API or the ANSI C standard library. Intel provides a routine to access the SPI hardware that is specific to the SPI core.

5.4.1. Hardware Access Routines

Intel provides one access routine, `alt_avalon_spi_command()`, that provides general-purpose access to the SPI core that is configured as a host.

5.4.1.1. alt_avalon_spi_command()

Prototype:	`int alt_avalon_spi_command(alt_u32 base, alt_u32 slave, alt_u32 write_length, const alt_u8* wdata, alt_u32 read_length, alt_u8* read_data, alt_u32 flags)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<altera_avalon_spi.h>
Description:	This function performs a control sequence on the SPI bus. It supports only SPI hosts with data width less than or equal to 8 bits. A single call to this function writes a data buffer of arbitrary length to the `mosi` port, and then reads back an arbitrary amount of data from the `miso` port. The function performs the following actions: (1) Asserts the agent select output for the specified agent. The first agent select output is 0. (2) Transmits write_length bytes of data from wdata through the SPI interface, discarding the incoming data on the `miso` port. (3) Reads read_length bytes of data and stores the data into the buffer pointed to by read_data. The `mosi` port is set to zero during the read transaction. (4) De-asserts the agent select output, unless the flags field contains the value ALT_AVALON_SPI_COMMAND_MERGE. If you want to transmit from scattered buffers, call the function multiple times and specify the merge flag on all the accesses except the last. To access the SPI bus from more than one thread, you must use a semaphore or mutex to ensure that only one thread is executing within this function at any time.
Returns:	The number of bytes stored in the read_data buffer.

5.4.2. Software Files

The core is accompanied by the following software files. These files provide a low-level interface to the hardware.

- `altera_avalon_spi.h`—This file defines the core's register map, providing symbolic constants to access the low-level hardware.
- `altera_avalon_spi.c`—This file implements low-level routines to access the hardware.

5.4.3. Register Map

An Avalon-MM host peripheral controls and communicates with the core via the six 32-bit registers, shown in below in the **Register Map for SPI Host Device** figure. The table assumes an n-bit data width for `rxdata` and `txdata`.

Table 18. Register Map for SPI Host Device

Internal Address	Register Name	Type [R/W]	32-11	10	9	8	7	6	5	4	3	2-0
0	`rxdata` (3)	R	RXDATA (n-1..0)									
1	`txdata` (3)	W	TXDATA (n-1..0)									
2	`status` (1)	R/W			EOP	E	RRDY	TRDY	TMT	TOE	ROE	
3	`control`	R/W		SSO (2)	IEOP	IE	IRRDY	ITRDY		ITOE	IROE	
4	Reserved	—										
5	`slaveselect` (2)	R/W	Agent Select Mask									
6	`eop_value`(3)	R/W	End of Packet Value (n-1..0)									

Reading undefined bits returns an undefined value. Writing to undefined bits has no effect.

5.4.3.1. rxdata Register

A host peripheral reads received data from the `rxdata` register. When the receive shift register receives a full n bits of data, the `status` register's RRDY bit is set to 1 and the data is transferred into the `rxdata` register. Reading the `rxdata` register clears the RRDY bit. Writing to the `rxdata` register has no effect.

New data is always transferred into the `rxdata` register, whether or not the previous data was retrieved. If RRDY is 1 when data is transferred into the `rxdata` register (that is, the previous data was not retrieved), a receive-overrun error occurs and the `status` register's ROE bit is set to 1. In this case, the contents of `rxdata` are undefined.

5.4.3.2. txdata Register

A host peripheral writes data to be transmitted into the `txdata` register. When the `status` register's TRDY bit is 1, it indicates that the `txdata` register is ready for new data. The TRDY bit is set to 0 whenever the `txdata` register is written. The TRDY bit is set to 1 after data is transferred from the `txdata` register into the transmitter shift register, which readies the `txdata` holding register to receive new data.

---

(1) A write operation to the `status` register clears the ROE, TOE, and E bits.

(2) Present only in host mode.

(3) Bits 31 to n are undefined when n is less than 32.

A host peripheral should not write to the txdata register until the transmitter is ready for new data. If TRDY is 0 and a host peripheral writes new data to the txdata register, a transmit-overrun error occurs and the status register's TOE bit is set to 1. In this case, the new data is ignored, and the content of txdata remains unchanged.

As an example, assume that the SPI core is idle (that is, the txdata register and transmit shift register are empty), when a CPU writes a data value into the txdata holding register. The TRDY bit is set to 0 momentarily, but after the data in txdata is transferred into the transmitter shift register, TRDY returns to 1. The CPU writes a second data value into the txdata register, and again the TRDY bit is set to 0. This time the shift register is still busy transferring the original data value, so the TRDY bit remains at 0 until the shift operation completes. When the operation completes, the second data value is transferred into the transmitter shift register and the TRDY bit is again set to 1.

5.4.3.3. status Register

The status register consists of bits that indicate status conditions in the SPI core. Each bit is associated with a corresponding interrupt-enable bit in the control register, as discussed in the **Control Register** section. A host peripheral can read status at any time without changing the value of any bits. Writing status does clear the ROE, TOE and E bits.

Table 19. status Register Bits

#	Name	Description
3	ROE	**Receive-overrun error** The ROE bit is set to 1 if new data is received while the rxdata register is full (that is, while the RRDY bit is 1). In this case, the new data overwrites the old. Writing to the status register clears the ROE bit to 0.
4	TOE	**Transmitter-overrun error** The TOE bit is set to 1 if new data is written to the txdata register while it is still full (that is, while the TRDY bit is 0). In this case, the new data is ignored. Writing to the status register clears the TOE bit to 0.
5	TMT	**Transmitter shift-register empty** In host mode, the TMT bit is set to 0 when a transaction is in progress and set to 1 when the shift register is empty. In agent mode, the TMT bit is set to 0 when the agent is selected (SS_n is low) or when the SPI Agent register interface is not ready to receive data.
6	TRDY	**Transmitter ready** The TRDY bit is set to 1 when the txdata register is empty.
7	RRDY	**Receiver ready** The RRDY bit is set to 1 when the rxdata register is full.
8	E	**Error** The E bit is the logical OR of the TOE and ROE bits. This is a convenience for the programmer to detect error conditions. Writing to the status register clears the E bit to 0.
9	EOP	**End of Packet** The EOP bit is set when the End of Packet condition is detected. The End of Packet condition is detected when either the read data of the rxdata register or the write data to the txdata register is matching the content of the eop_value register.

5.4.3.4. control Register

The `control` register consists of data bits to control the SPI core's operation. A host peripheral can read `control` at any time without changing the value of any bits.

Most bits (`IROE`, `ITOE`, `ITRDY`, `IRRDY`, and `IE`) in the `control` register control interrupts for status conditions represented in the `status` register. For example, bit 1 of `status` is ROE (receiver-overrun error), and bit 1 of control is `IROE`, which enables interrupts for the ROE condition. The SPI core asserts an interrupt request when the corresponding bits in `status` and `control` are both 1.

Table 20. control Register Bits

#	Name	Description
3	IROE	Setting IROE to 1 enables interrupts for receive-overrun errors.
4	ITOE	Setting ITOE to 1 enables interrupts for transmitter-overrun errors.
6	ITRDY	Setting ITRDY to 1 enables interrupts for the transmitter ready condition.
7	IRRDY	Setting IRRDY to 1 enables interrupts for the receiver ready condition.
8	IE	Setting IE to 1 enables interrupts for any error condition.
9	IEOP	Setting IEOP to 1 enables interrupts for the End of Packet condition.
10	SSO	Setting SSO to 1 forces the SPI core to drive its ss_n outputs, regardless of whether a serial shift operation is in progress or not. The slaveselect register controls which ss_n outputs are asserted. SSO can be used to transmit or receive data of arbitrary size, for example, greater than 32 bits.

After reset, all bits of the `control` register are set to 0. All interrupts are disabled and no ss_n signals are asserted.

5.4.3.5. slaveselect Register

The `slaveselect` register is a bit mask for the ss_n signals driven by an SPI host. During a serial shift operation, the SPI host selects only the agent device(s) specified in the `slaveselect` register.

The `slaveselect` register is only present when the SPI core is configured in host mode. There is one bit in `slaveselect` for each ss_n output, as specified by the designer at system generation time. The `slaveselect` register value is only updated either at the beginning of the actual SPI transmission or when the SSO bit is set from 0 to 1.

A host peripheral can set multiple bits of `slaveselect` simultaneously, causing the SPI host to simultaneously select multiple agent devices as it performs a transaction. For example, to enable communication with agent devices 1, 5, and 6, set bits 1, 5, and 6 of `slaveselect`. However, consideration is necessary to avoid signal contention between multiple agents on their `miso` outputs.

Upon reset, bit 0 is set to 1, and all other bits are cleared to 0. Thus, after a device reset, agent device 0 is automatically selected.

5.4.3.6. end of packet value Register

The end of packet value register allows you to specify the value of the SPI data word. The SPI data word acts as the end of packet word.

5.5. Example Test Code

```
#include "alt_types.h"
#include "sys/alt_stdio.h"
#include "io.h"
#include "system.h"
#include "sys/alt_cache.h"
#include "altera_avalon_spi.h"
#include "altera_avalon_spi_regs.h"
#include "sys/alt_irq.h"

//This is the ISR that runs when the SPI Slave receives data

static void spi_rx_isr(void* isr_context){

        alt_printf("ISR :) %x \n" ,
IORD_ALTERA_AVALON_SPI_RXDATA(SPI_SLAVE_BASE));

        //This resets the IRQ flag. Otherwise the IRQ will continuously run.
        IOWR_ALTERA_AVALON_SPI_STATUS(SPI_SLAVE_BASE, 0x0);
}

int main()
{
  alt_printf("Hello from Nios II!\n");

  int return_code,ret;
  char spi_command_string_tx[10] = "$HELLOABC*";

  char spi_command_string_rx[10] = "$HELLOABC*";

  //This registers the Slave IRQ with NIOS

  ret =
alt_ic_isr_register(SPI_SLAVE_IRQ_INTERRUPT_CONTROLLER_ID,SPI_SLAVE_IRQ,spi_rx_i
sr,(void *)spi_command_string_tx,0x0);
  alt_printf("IRQ register return %x \n", ret);

  //You need to enable the IRQ in the IP core control register as well.

IOWR_ALTERA_AVALON_SPI_CONTROL(SPI_SLAVE_BASE,ALTERA_AVALON_SPI_CONTROL_SSO_MSK
| ALTERA_AVALON_SPI_CONTROL_IRRDY_MSK);

  //Just calling the ISR to see if the function is OK.
  spi_rx_isr(NULL);

  return_code = alt_avalon_spi_command(SPI_MASTER_BASE,0 ,
                               1, spi_command_string_tx,
                               0, spi_command_string_rx,
                               0);

  return_code = alt_avalon_spi_command(SPI_MASTER_BASE,0 ,
                                 1, &spi_command_string_tx[1],
                                 0, spi_command_string_rx,
                                 0);

  return_code = alt_avalon_spi_command(SPI_MASTER_BASE,0 ,
                                 1, &spi_command_string_tx[2],
                                 0, spi_command_string_rx,
                                 0);

  return_code = alt_avalon_spi_command(SPI_MASTER_BASE,0 ,
                                 1, &spi_command_string_tx[3],
```

```
                                0, spi_command_string_rx,
                                0);

  if(return_code < 0)
                alt_printf("ERROR SPI TX RET = %x \n" , return_code);

  alt_printf("Transmit done. RET = %x spi_rx %x
\n",return_code,spi_command_string_rx[0]);

  //RX is done via interrupts.
  alt_printf("Rx done \n");
  return 0;

}
```

5.6. SPI Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2021.03.29	21.1	Clarified the SDATA to IP clock sampling requirement in section: *SPI Clock (sclk) Rate*.
2019.12.16	19.4	Added the following new sections: • *Synchronizer Stages* • *Example Test Code*
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
June 2016	2016.06.17	Updates: • Removed content regarding Avalon-MM flow control • Table 18 on page 51: `eop_value` added • Table 19 on page 52: `EOP` added • Table 20 on page 53: `IEOP` added • end of packet value Register on page 54: New topic
December 2010	v10.1.0	Removed the "Device Support", "Instantiating the Core in SOPC Builder", and "Referenced Documents" sections.
July 2010	v10.0.0	No change from previous release.
November 2009	v9.1.0	Revised register width in transmitter logic and receiver logic. Added description on the disable flow control option. Added R/W column in Table 8–3 .
March 2009	v9.0.0	No change from previous release.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. Updated the width of the parameters and signals from 16 to 32.
May 2008	v8.0.0	Updated the description of the `TMT` bit.

6. SPI Agent/JTAG to Avalon Host Bridge Cores

6.1. Core Overview

The SPI Agent to Avalon Host Bridge and the JTAG to Avalon Host Bridge cores provide a connection between host systems and Platform Designer systems via the respective physical interfaces. Host systems can initiate Avalon Memory-Mapped (Avalon-MM) transactions by sending encoded streams of bytes via the cores' physical interfaces. The cores support reads and writes, but not burst transactions.

This IP core supports the Motorola SPI Protocol, using the SPI clock phase bit, `CPHA = 1`, and SPI clock polarity bit, `CPOL = 0` mode.

Note: This IP core supports Intel eASIC N5X devices.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

6.2. Functional Description

Figure 15. System with a SPI Agent to Avalon Host Bridge Core

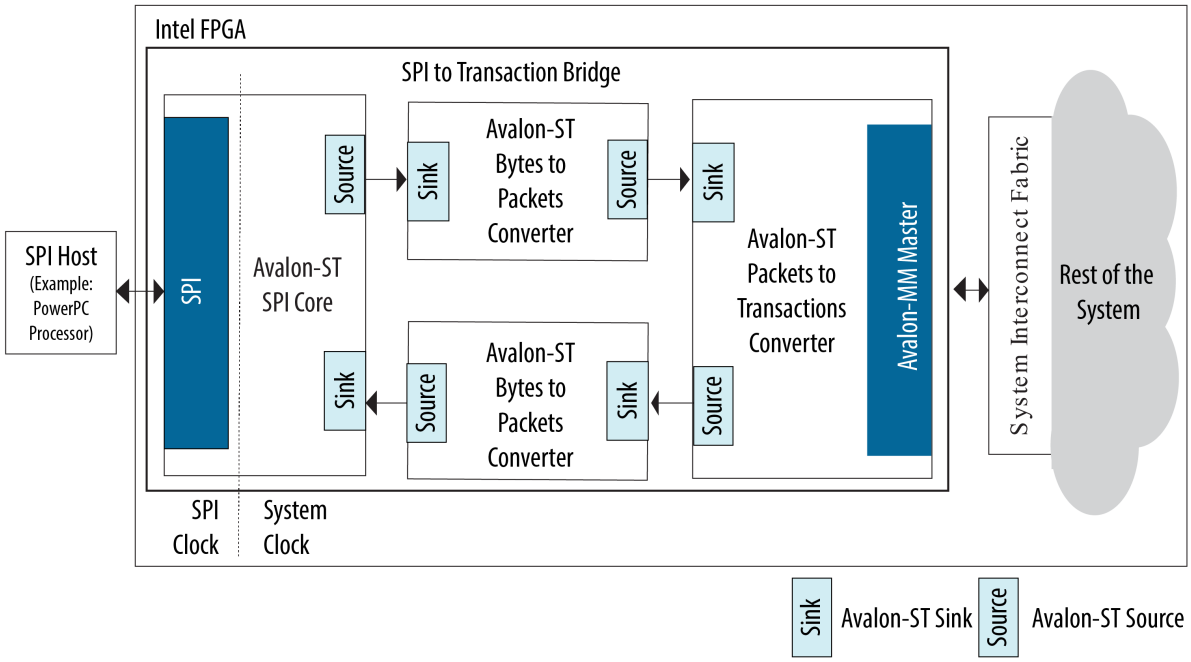
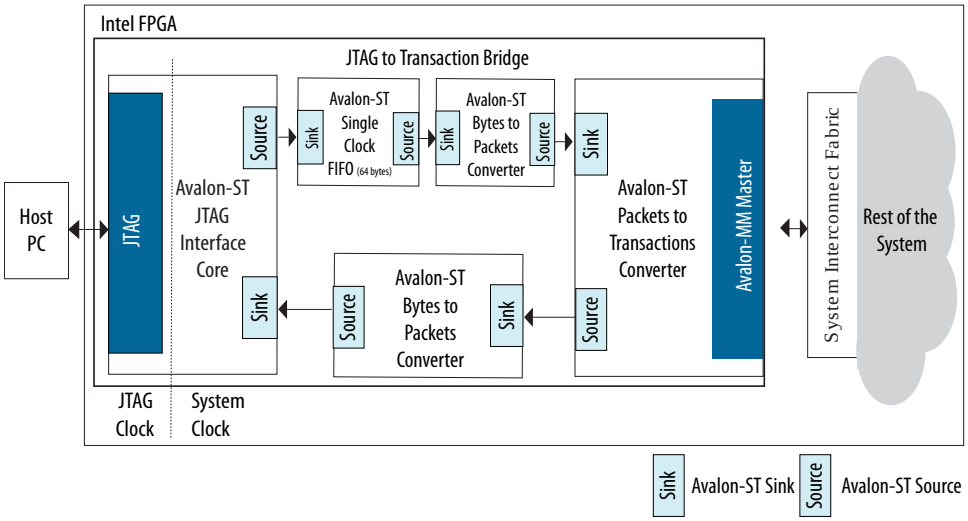


Figure 16. System with a JTAG to Avalon Host Bridge Core

Note: System clock must be at least 2X faster than the JTAG clock.



The SPI Agent to Avalon Host Bridge and the JTAG to Avalon Host Bridge cores accept encoded streams of bytes with transaction data on their respective physical interfaces and initiate Avalon-MM transactions on their Avalon-MM interfaces. Each bridge consists of the following cores, which are available as stand-alone components in Platform Designer:

- **Avalon-ST Serial Peripheral Interface and Avalon-ST JTAG Interface**—Accepts incoming data in bits and packs them into bytes.
- **Avalon-ST Bytes to Packets Converter**—Transforms packets into encoded stream of bytes, and a likewise encoded stream of bytes into packets.
- **Avalon-ST Packets to Transactions Converter**—Transforms packets with data encoded according to a specific protocol into Avalon-MM transactions, and encodes the responses into packets using the same protocol.
- **Avalon-ST Single Clock FIFO**—Buffers data from the Avalon-ST JTAG Interface core. The FIFO is only used in the JTAG to Avalon Host Bridge.

  For the bridges to successfully transform the incoming streams of bytes to Avalon-MM transactions, the streams of bytes must be constructed according to the protocols used by the cores.

Note: When you connect the JTAG Avalon Host Bridge component to a agent that back-pressures the host interface on this component, then using the System Console `master_write_from_file` command may result in data loss at the host interface or hung command in System Console.

Figure 17. Bits to Avalon-MM Transaction (Write)

The following example shows how a bytestream changes as it is transferred through the different layers in the bridges for write transaction.

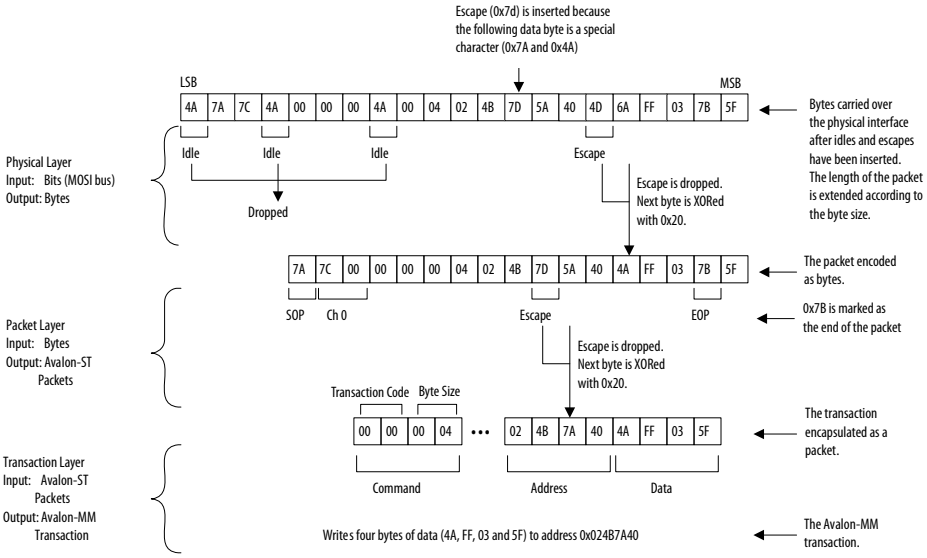


Figure 18. Write Response

After sending a write transaction packet through MOSI bus, the host has to initiate 8 bytes of IDLE transaction on the MOSI bus in order to get the write response from the MISO bus. The following figure shows the write response transaction that constructed by the bridge in the MISO bus. The most significant bit of the command is inversed.

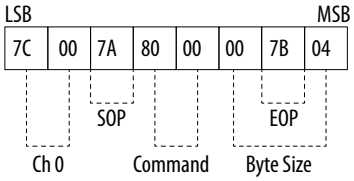


Figure 19. Bits to Avalon-MM Transaction (Read)

The following figure shows how a bytestream changes as it is transferred through the different layers in the bridges for a read transaction.

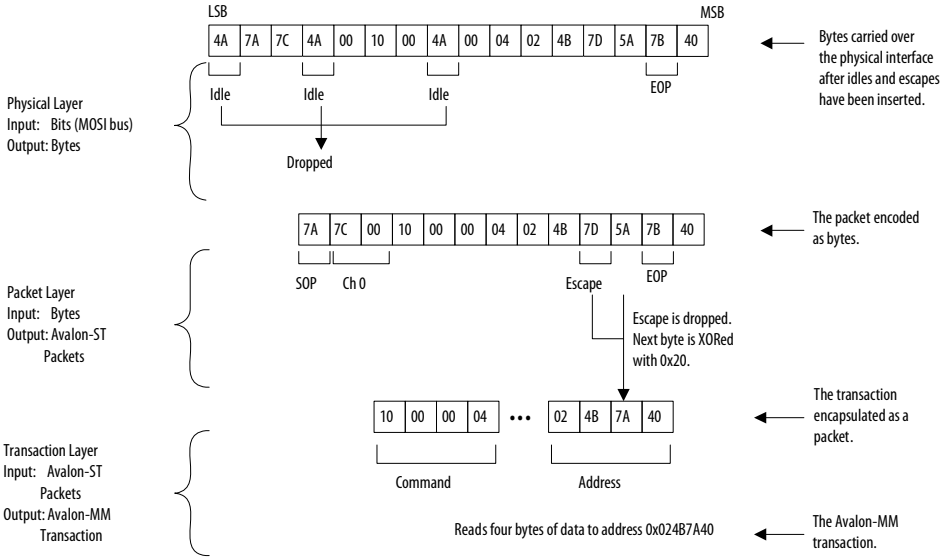
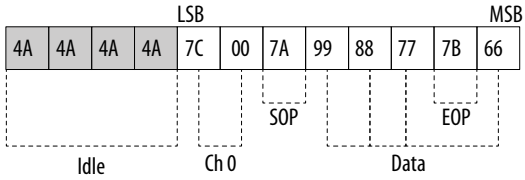


Figure 20. Read Response

After sending a read transaction through MOSI bus, the host has to initiate IDLE transaction on the MOSI bus to get the read response from the MISO bus. There is a possibility that the Avalon agent device is yet to return the read data, therefore the bridge returns 0x4A until read data is received. When read data is received by the bridge, it sends channel byte as the first byte followed by the SOP and data byte. The following figure shows the read response transaction that constructed by the bridge in the MISO bus.



Related Information

- Avalon-ST Serial Peripheral Interface Core on page 37
- Avalon-ST Bytes to Packets and Packets to Bytes Converter Cores on page 531
- Avalon Packets to Transactions Converter Core on page 520
- Avalon-ST Single-Clock and Dual-Clock FIFO Cores on page 31

6.3. Parameters

For the SPI Agent to Avalon Host Bridge core, the parameter **Number of synchronizer stages: Depth** allows you to specify the length of synchronization register chains. These register chains are used when a metastable event is likely to occur and the length specified determines the meantime before failure. The register chain length, however, affects the latency of the core.

For more information on metastability in Intel FPGA devices, refer to AN 42: Metastability in Intel FPGA devices.

For more information on metastability analysis and synchronization register chains, refer to the Area and Timing Optimization chapter in volume 2 of the *Intel Quartus Prime Handbook*.

6.4. SPI Agent/JTAG to Avalon Host Bridge Cores Revision History

Document Version	Intel Quartus Prime Version	Changes
2021.03.29	21.1	Added support for Intel eASIC N5X devices.
2019.04.01	19.1	Updated information about read and write transactions in *Functional Description*.
2018.05.07	18.0	• Corrected the *Figure: Bits to Avalon Memory-Mapped Interface Transaction (Read)*. • Implemented editorial enhancements.

Date	Version	Changes
May 2017	2017.05.08	Read operation added: Figure 19 on page 60
July 2014	2014.07.24	Removed mention of SOPC Builder, updated to Platform Designer
December 2010	v10.1.0	Removed the "Device Support", "Instantiating the Core in SOPC Builder", and "Referenced Documents" sections.
July 2010	v10.0.0	No change from previous release.
November 2009	v9.1.0	No change from previous release.
March 2009	v9.0.0	Added description of a new parameter **Number of synchronizer stages: Depth**.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	v8.0.0	Initial release.

7. Intel eSPI Agent Core

The Intel Enhanced SPI (eSPI) agent core allows you to communicate with an eSPI host such as the Platform Controller Hub (PCH) on a mother board using Intel MAX 10 device.

Related Information

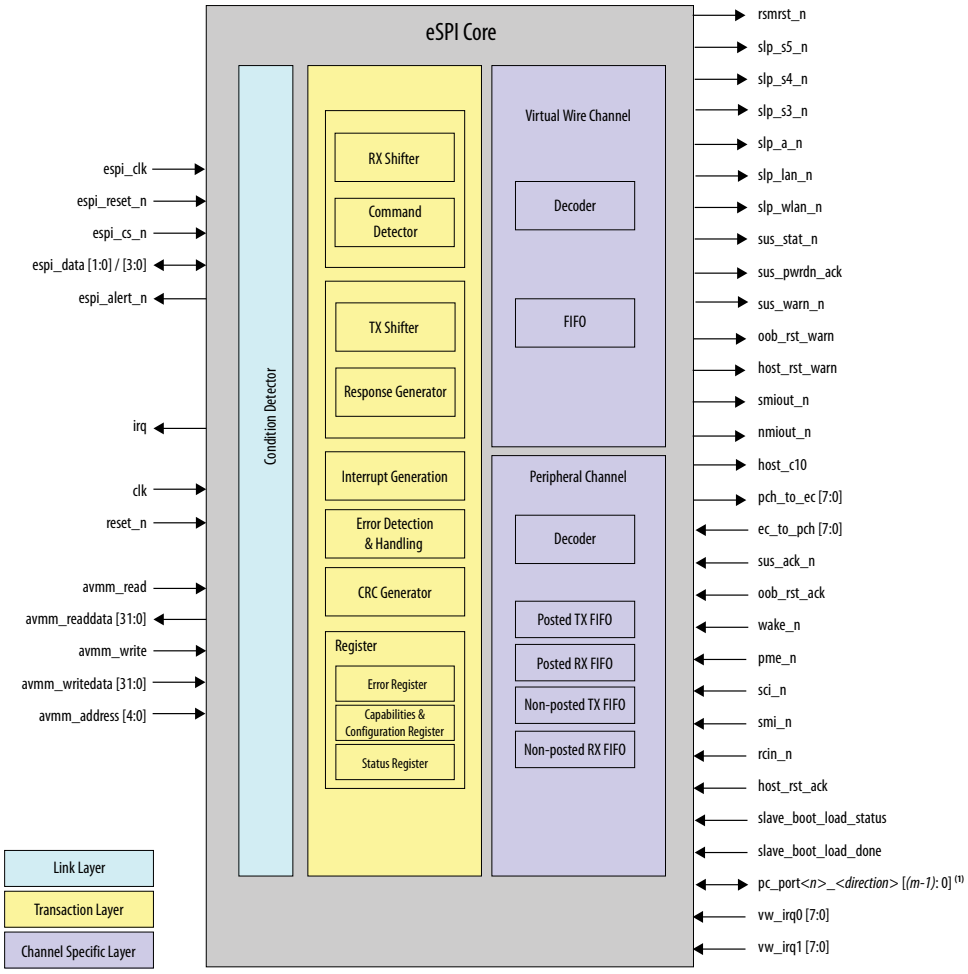
Enhanced Serial Peripheral Interface (eSPI) Base Specification
Provides more information about the architecture of eSPI bus interface for both client and server platforms.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

7.1. Functional Description

Figure 21. Block Diagram



(1) Configurable pc_port to input and output, support port00 to portA0. For example: pc_port80_in or pc_port80_out. Refer to the Interface Signal section for more details.

7.1.1. Link Layer

In the Link Layer, the Condition Detector block shifts the serial data bus in (receive) and out (transmit) in eSPI clock domain. The input serial data is translated into parallel form and sent to transaction layer. The parallel data bus from the transaction layer is translated into serial form in the condition detector and sent out as the eSPI output data.

During the single I/O mode, the `espi_data[1:0]` I/O pins are unidirectional to form an unidirectional data bus. Data is driven using `espi_data[0]` during the command phase, and `espi_data[1]` the during response phase. The eSPI agent is required to tri-state `espi_data[1]` during command phase as `espi_data[1]` can be driven by eSPI host such as when initiating an In-Band Reset command.

During the dual I/O mode, the `espi_data[1:0]` I/O pins are bi-directional to form a bi-directional data bus. All the command and response phases are transferred over the two bidirectional pins at the same time, which effectively doubles the transfer rate than that of the single I/O mode.

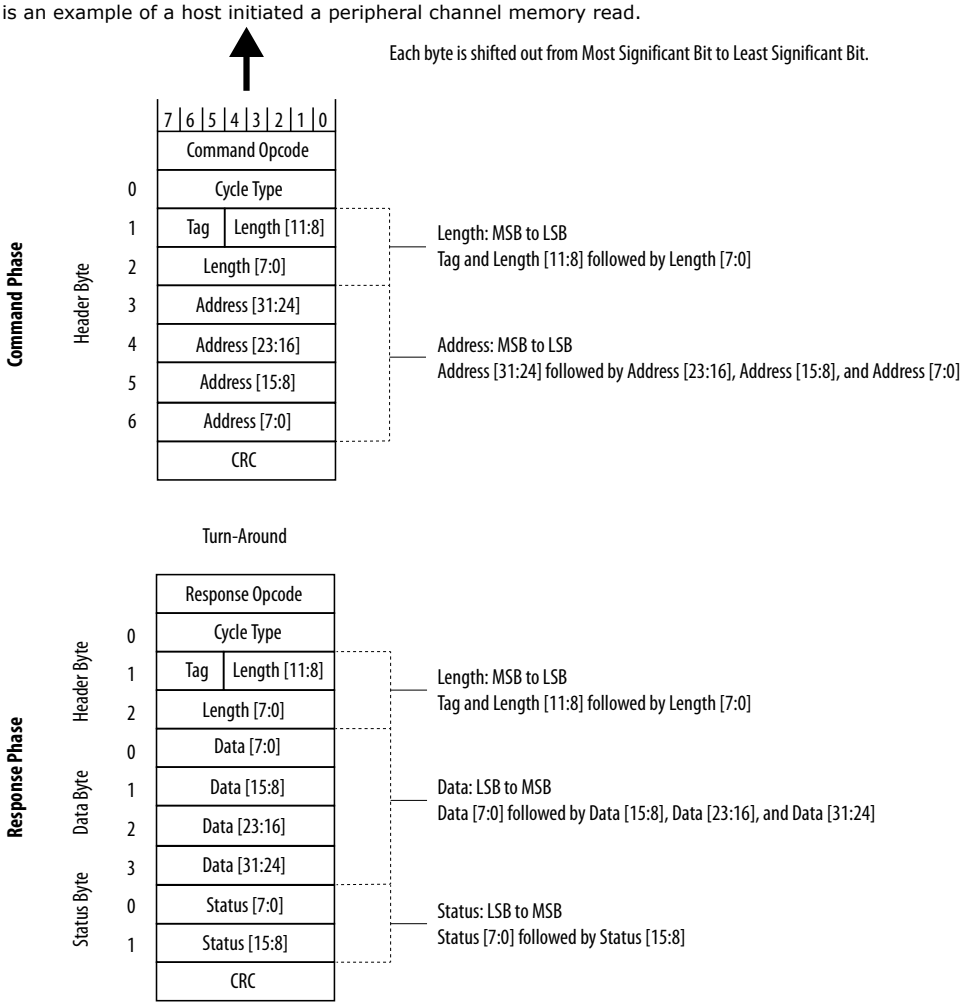
During the quad I/O mode, the `espi_data[3:0]` I/O pins are bi-directional data bus. All the command and response phases are transferred over the four bi-directional pins at the same time, which effectively doubles the transfer rate than that of the dual I/O mode.

During eSPI transaction, each field is shifted in a defined order. For multi-bytes field, the shifting order is as following:

- Header (length and address): Most Significant Byte (MSB) to Least Significant Byte (LSB)
- Data: LSB to MSB
- Status: LSB to MSB

Figure 22. Byte Ordering on the eSPI Bus

This is an example of a host initiated a peripheral channel memory read.



7.1.2. Transaction Layer

The RX Shifter block in the transaction layer gets the parallel data bus from the link layer. It identifies the fields of an eSPI transaction such as Header, Data, or Status and sends it to the Command Detector block for decoding. On the transmit side, the Response Generator block gathers information about the eSPI transaction. Thereafter, the TX shifter block sends out the transaction as per the field order to the link layer.

The transaction layer also contains an Error Detection and Handling block. There is no error correction capability or hardware recovery mechanism defined for the eSPI bus. The eSPI agent core categorize and handles the errors detected on the eSPI bus as following:

Table 21. Error Detection and Handling Categories

Error Condition	eSPI Agent Core Response and Handling
Invalid command opcode	Command is discarded. eSPI agent core response with NO_RESPONSE opcode.
Invalid cycle type	Command is discarded. eSPI agent core response with NO_RESPONSE opcode.
Command phase CRC error	Command is discarded. eSPI agent core response with NO_RESPONSE opcode.
Unexpected espi_cs_n de-assertion	eSPI agent core tri-state the data bus after espi_cs_n is deasserted and follows tSHQZ rule. It also triggers CRC error during response phase only if CRC checking is enabled.
Protocol error: • PUT without FREE • GET without AVAILABLE	Command is discarded. eSPI agent core response with FATAL_ERROR opcode.
Malformed packet during command phase: • Peripheral Channel: — Payload length > Maximum Payload Size — Read Request Size > Maximum Read Request Size — (Address + Length) > 4Kb • Virtual Wire Channel: — Count > Maximum Virtual Wire Count	Command is discarded. eSPI agent core response with FATAL_ERROR opcode.
When PORT 00h to PORT 100h only supports PC_CHANNEL IO SHORT command and the host issues IO READ/WRITE SHORT command to address outside of 00h to 100h.	Command is discarded. eSPI agent core response with NON_FATAL_ERROR opcode.

Besides the Error Detection and Handling block, there is a CRC-8 Generator block to protect the eSPI transaction packets. The command phase and response phase contain their respective CRC byte. For command phase, the CRC calculation includes all the bytes during the command phase such as the Command Opcode, Header and Data. For response phase, the CRC calculation includes all the bytes during the response phase such as the Response Opcode (except WAIT_STATE), Header, Data and Status. The CRC checking is disabled by default. To enable CRC checking, set the CRC Checking Enable bit using the SET_CONFIGURATION command.

7.1.3. Channel Specific Layer

Currently, the eSPI agent core only supports Channel 0 and Channel 1. However, the eSPI agent core is designed to support future expansion for other channel specification such as OOB and Flash Access Channel.

7.1.3.1. Peripheral Channel

The peripheral channel allows you to communicate between the eSPI host and the eSPI endpoints located at the agent side (example: PORT80). To reset the channel, use the platform reset (PLTRST_n VW).

You can enable the peripheral IO access and configure the port width and direction using the Platform Designer. The eSPI agent core allocates address range from 00h to A0h to excess the peripheral IO. See *Table: Peripheral IO Port Configuration* for more details. By default, the pc_port80 has 8-bit data width. Each address location can be configured as 8-bit wide, 16-bit wide or 32-bit wide.

When you set an IO port to 8-bit wide, you must access the port using PUT_IORD_SHORT 1 byte/PUT_IOWR_SHORT 1 byte command. When you set an IO port to 16-bit wide, you must access the port using PUT_IORD_SHORT 2 byte/ PUT_IOWR_SHORT 2 byte command. When you set an IO port to 32-bit wide, you must access the port using PUT_IORD_SHORT 4 byte/PUT_IOWR_SHORT 4 byte command.

PUT_PC or PUT_NP loads packets into the FIFO while GET_PC or GET_NP flushes out packets from the FIFO. Decoder decodes address bytes from the header and sends data out to the corresponding output port.

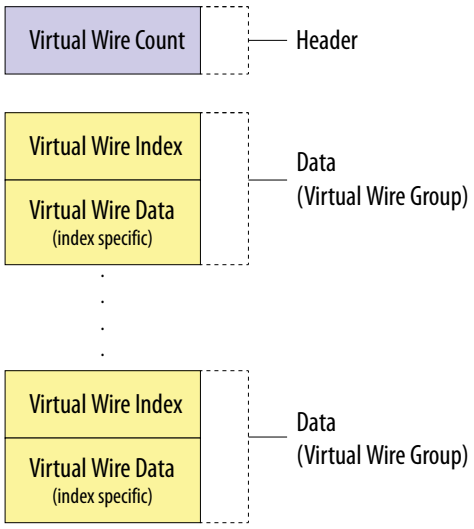
Table 22. Peripheral IO Port Configuration

Address	Data Width	Port Name	Port Direction	Enable
00h	8/16/32	pc_port00	Input/Output	Yes/No
10h	8/16/32	pc_port10	Input/Output	Yes/No
20h	8/16/32	pc_port20	Input/Output	Yes/No
30h	8/16/32	pc_port30	Input/Output	Yes/No
40h	8/16/32	pc_port40	Input/Output	Yes/No
50h	8/16/32	pc_port50	Input/Output	Yes/No
60h	8/16/32	pc_port60	Input/Output	Yes/No
70h	8/16/32	pc_port70	Input/Output	Yes/No
80h	8/16/32	pc_port80	Input/Output	Yes/No
90h	8/16/32	pc_port90	Input/Output	Yes/No
A0h	8/16/32	pc_portA0	Input/Output	Yes/No

7.1.3.2. Virtual Wire Channel

The virtual wire (VW) channel allows you to communicate the state of the GPIO tunneled through eSPI bus as in-band messages. The command phase consists of a command opcode, virtual wire count, virtual wire index, virtual wire data and CRC.

Figure 23. Virtual Wire Packet



The virtual wire count indicates the number of virtual wire groups communicated by the packet. It is followed by one or more virtual wire groups. Each virtual wire group consists of 2 bytes, namely the virtual wire index and the virtual wire data. The eSPI agent core supports System Event, and Server Platform Specific virtual wire index.

Table 23. System Event Virtual Wire Index

Virtual Wire Index	Direction	Virtual Wire Data Byte								Reset
		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	
2h	Host to Agent	Valid bit for Bit 3- Bit 0				Reserved	SLP_S5_n	SLP_S4_n	SLP_S3_n	rsmrst_n
3h	Host to Agent	Valid bit for Bit 3- Bit 0				Reserved	OOB_RST_WARN	PLTRST_n	SUS_STAT_n	espi_reset_n
4h	Agent to Host	Valid bit for Bit 3- Bit 0				PME_n	WAKE_n	Reserved	OOB_RST_ACK	espi_reset_n
5h	Agent to Host	Valid bit for Bit 3- Bit 0				SLAVE_BOOT_LOAD_STATUS	-	-	SLAVE_BOOT_LOAD_DONE	espi_reset_n
6h	Agent to Host	Valid bit for Bit 3- Bit 0				HOST_RST_ACK	RCIN_n	SMI_n	SCI_n	PLTRST_n VW
7h	Host to Agent	Valid bit for Bit 3- Bit 0				Reserved	NMIOUT_n	SMIOUT_n	HOST_RST_WARN	PLTRST_n VW

Table 24. Server Platform Specific Virtual Wire Index

Virtual Wire Index	Direction	Virtual Wire Data Byte								Reset
		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	
40h	Agent to Host	Valid bit for Bit 3- Bit 0				Reserved	Reserved	Reserved	SUS_ACK_n	espi_reset_n
41h	Host to Agent	Valid bit for Bit 3- Bit 0				SLP_A_n	Reserved	SUS_PWRDN_ACK	SUS_WARN_n	espi_reset_n
42h	Host to Agent	Valid bit for Bit 3- Bit 0				Reserved	Reserved	SLP_WLAN_n	SLP_LAN_n	rsmrst_n
43h	Host to Agent	Valid bit for Bit 3- Bit 0				PCH_TO_EC_3	PCH_TO_EC_2	PCH_TO_EC_1	PCH_TO_EC_0	espi_reset_n
44h	Host to Agent	Valid bit for Bit 3- Bit 0				PCH_TO_EC_7	PCH_TO_EC_6	PCH_TO_EC_5	PCH_TO_EC_4	espi_reset_n
45h	Agent to Host	Valid bit for Bit 3- Bit 0				EC_TO_PCH_3	EC_TO_PCH_2	EC_TO_PCH_1	EC_TO_PCH_0	espi_reset_n
46h	Agent to Host	Valid bit for Bit 3- Bit 0				EC_TO_PCH_7	EC_TO_PCH_6	EC_TO_PCH_5	EC_TO_PCH_4	espi_reset_n
47h	Host to Agent	Valid bit for Bit 3- Bit 0				Reserved	Reserved	Reserved	HOST_C10	PLTRST_n VW

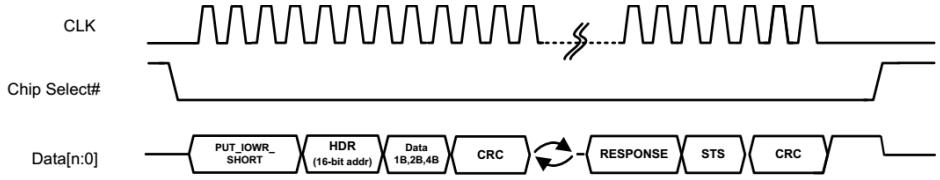
7.1.4. Port80 Implementation

To send a Port80 message, the eSPI host:

1. sends `PUT_IOWR_SHORT 1 byte` as the command
2. 80h as the address bytes in header
3. followed by a byte of data and CRC

The port80[7:0] gets 8-bit data. After, 2 clock cycles of turn around time, the eSPI agent core sends back ACCEPT as the response code, and followed by the status register value and CRC.

Figure 24. eSPI Host Initiated Short IO Write Transaction



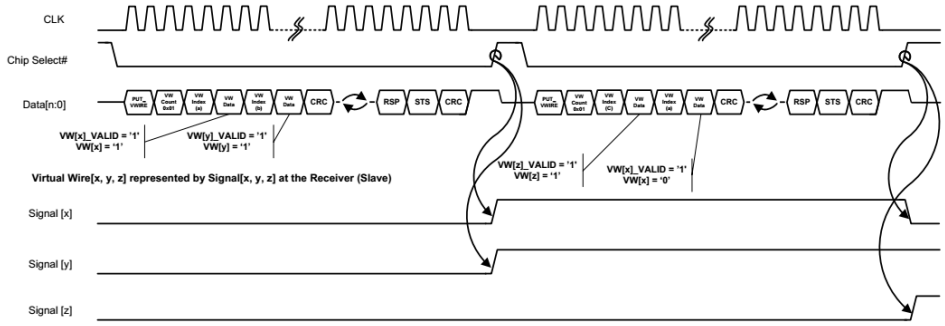
7.1.5. VW message to Physical Port Implementation

To assert the output ports that connect to the Virtual Wire channel, the eSPI host:

1. sends PUT_VWIRE as the command
2. assign a virtual wire count value
3. assign the virtual wire index of the corresponding output port
4. followed by a byte of data and CRC

For example, when the eSPI host is asserting SLP_S3_n, the virtual wire index is 2h and data byte is 8'b00010110. The virtual wire tunneled through eSPI takes effect at the receiver only when the espi_cs_n is deasserted.

Figure 25. Virtual Wire Transaction



When the logical state of the output ports which connect to the Virtual Wire channel changes, the new state must be communicated to the eSPI host. In this case, the eSPI agent core initiates an Alert event to the eSPI host. Then, eSPI host triggers a GET_STATUS transaction, followed by a GET_VWIRE transaction.

7.1.6. Avalon Memory-Mapped Interface Interface Settings

Table 25. Avalon Memory-Mapped Interface Interface Settings

Setting Field	Value
readLatency	2
readWaitTime	0
writeWaitTime	0
addressUnits	WORDS
bitsPerSymbol	8
maximumPendingReadTransactions	0
maximumPendingWriteTransactions	0
burstOnBurstBoundariesOnly	false

7.2. Resource Utilization

Table 26. eSPI Agent Core Resource Utilization

Device	Resource	Value
Intel MAX 10	F_{MAX}	150 MHz(4)
	Logic cells	1586
	Dedicated logic registers	813
	M9Ks	5

7.3. IP Parameters

Table 27. Parameter List

You can configure these parameter using the Platform Designer.

Parameter Name	Default Value	Description
eSPI Mode of Operation	Single I/O	Allows you to set I/O configuration. Available options: • Single I/O • Single and Dual I/O • Single and Quad I/O • Single, Dual, and Quad I/O
Frequency of Operation	20MHz	Allows you to set the operational frequency. Available options: • 20MHz • 25MHz • 33MHz • 50MHz • 66MHz(5)
Channel Supported	Peripheral Channel	Allows you to set the channel support. Available option: • Virtual Wire Channel • Peripheral Channel • All Channel
Peripheral Channel Maximum Payload Size Supported	64 bytes	Allows you to set the maximum payload size for peripheral channel. Available options: • 64 bytes
Peripheral Channel Maximum Read Request Size Supported	64 bytes	Allows you to set the maximum read request size for peripheral channel. Available options: • 64 bytes
Maximum Virtual Wire Count Supported (0-based)	0x07	Allows you to set the maximum virtual wire count. Legal values:

continued...

(4) The clock ratio between eSPI clock and the IP clock must be x3. Intel recommends to run eSPI clock of 66MHz in Intel Arria 10 device and with IP clock above 200MHz. Currently, the eSPI clock of 66MHz is not supported in Intel MAX 10 devices.

(5) The clock ratio between eSPI clock and the IP clock must be x3. Intel recommends to run eSPI clock of 66MHz in Intel Arria 10 device and with IP clock above 200MHz. Currently, the eSPI clock of 66MHz is not supported in Intel MAX 10 devices.

Parameter Name	Default Value	Description
		• 0x07: 8 count • 0x08: 9 count • 0x09: 10 count • 0x0A: 11 count • 0x0B: 12 count • 0x0C: 13 count • 0x0D: 14 count • 0x0E: 15 count • 0x0F: 16 count

7.4. Interface Signals

Table 28. Interface Signals

Signal Names	Width (bit)	Direction	Description
Clock Interface			
clk	1	Input	Input clock signal.
Reset Interface			
reset_n	1	Input	Synchronous reset signal.
Avalon-MM Agent Interface(6)			
avmm_read	1	Input	Use this signal to enable read from the status register, error register, posted RX fifo or non-posted RX fifo.
avmm_readdata[31:0]	32	Output	Use this signal to read data from the status register , error register, posted RX fifo or non-posted RX fifo.
avmm_write	1	Input	Use this signal to enable write to posted TX fifo or non-posted TX fifo.
avmm_writedata[31:0]	32	Input	Use this signal to write data to posted TX fifo or non-posted TX fifo.
avmm_address[4:0]	5	Input	Avalon address determines address to access the respective register/fifo.
Interrupt Signal			
irq	1	Output	Interrupt signal reflects update to status register. It is also triggered when the error register bit is asserted.
eSPI Interface			
espi_clk	1	Input	eSPI serial clock signal. Frequency range: 20MHz to 66 MHz.
espi_reset_n	1	Input	eSPI reset signal.
espi_cs_n	1	Input	eSPI chip select signal.

continued...

(6) Use Avalon-MM interface to access the status register with Avalon address set to 00h and also the FIFO in the PC channel with Avalon address set to 04h.

Signal Names	Width (bit)	Direction	Description
espi_data[1:0]/[3:0]	2 or 4	Input/Output	eSPI bi-directional data bus. Data bus configuration is determined by the **eSPI Mode of Operation** parameter. • 2-bit data bus: — Single I/O — Single and Dual I/O • 4-bit data bus: — Single and Quad I/O — Single, Dual and Quad I/O
espi_alert_n	1	Output	eSPI alert signal.
Conduit			
slp_s5_n	1	Output	S5 sleep control signal is sent when the power to non-critical systems should be shut off in S5.
slp_s4_n	1	Output	S4 sleep control signal is sent when the power to non-critical systems should be shut off in S4.
slp_s3_n	1	Output	S3 sleep control signal is sent when the power to non-critical systems should be shut off in S3.
slp_a_n	1	Output	Use sleep A signal to support **ASW** devices that needs power in the SX platform when the Intel ME is on.
slp_lan_n	1	Output	LAN sub-system sleep control signal is sent when the power to external wired LAN PHY can be shut off.
slp_wlan_n	1	Output	Wireless LAN sub-system sleep control signal is sent when the power to external wireless LAN PHY can be shut off.
sus_stat_n	1	Output	Suspend status signal is sent when the system is about to enter a low power state.
sus_pwrdn_ack	1	Output	Suspend power down acknowledgement signal.
sus_warn_n	1	Output	Suspend warning signal.
oob_rst_warn	1	Output	Host sends this signal before the OOB processor is about to reset.
host_rst_warn	1	Output	Host sends this signal before the host is about to reset.
smiout_n	1	Output	Host sends this signal indicating the occurrence of SMI event.
nmiout_n	1	Output	Host sends this signal indicating the occurrence of NMI event.
host_c10	1	Output	Indicates that the host CPU has entered deep power down state C10 or deeper.
pch_to_ec[7:0]	8	Output	8 independent virtual wire placeholder from the platform controller hub (eSPI host) to the eSPI agent IP.
ec_to_pch[7:0]	8	Input	8 independent virtual wire placeholder from eSPI agent IP to the platform controller hub (eSPI host).

continued...

Signal Names	Width (bit)	Direction	Description
sus_ack_n	1	Input	Suspend acknowledgement signal.
oob_rst_ack	1	Input	OOB reset acknowledgement signal.
wake_n	1	Input	This signal wakes up host from Sx on any event. It can also wake up on LID switch or AC insertion event.
pme_n	1	Input	This signal wakes up host from Sx through PCI defined PME.
sci_n	1	Input	General purpose alert signal which results in OS invoking ACPI method.
smi_n	1	Input	General purpose alert signal which results in BIOS invoking SMI code.
rcin_n	1	Input	To request CPU reset on behalf of the keyboard controller.
host_rst_ack	1	Input	Host reset acknowledgement signal.
slave_boot_load_done	1	Input	Indicates the boot load completion.
slave_boot_load_status	1	Input	Indicates the boot load status.
pc_port<*n*>_<*direction*>[(*m-1*):0]	*m* = 8/16/32	Input/Output	Peripheral channel IO ports with configurable data width and direction. *n* = configurable value from 00 to A0. For example: pc_port80_out[15:0]
rsmrst_n	1	Input	This signal provides input reset to some of the virtual wire index.

7.5. Registers

7.5.1. Avalon-MM Interface Accessible Registers

Table 29. Avalon-MM Interface Accessible Register Map

Offset Address	Register Name	Access Type	Width (bits)
0x0h	Status Register	R	32
0x4h	Control Register	W	32
0x8h	Posted RX FIFO(7)	R	8
0xCh	Posted TX FIFO(8)	W	8

continued...

(7) eSPI host issue command such as PUT_PC , or PUT_MEMWR_SHORT to write data byte into this FIFO. To retrieve data from this FIFO, use Avalon read signal.

(8) Issue Avalon Write command to write response data into this FIFO.

Offset Address	Register Name	Access Type	Width (bits)
0x10h	Non-posted RX FIFO[9]	R	8
0x14h	Non-posted TX FIFO[8]	W	8
0x18h	Error Register	R/W1C	32

7.5.1.1. Status Register

Table 30. Status Register Bit Description

Bit	Type	Status Field	Description
0	R	PCRXFIFO_AVAIL	The following values indicates: • 1: Channel 0 Posted RXFIFO has a peripheral posted or completion header. And data up to maximum payload size is available for read. • 0: Channel 0 Posted RXFIFO is empty.
1	R	NPRXFIFO_AVAIL	The following values indicates: • 1: Channel 0 Non-posted RXFIFO has a peripheral non-posted header available for read. • 0: Channel 0 Non-posted RXFIFO is empty.
31:2	-	-	Reserved.

7.5.1.2. Control Register

Table 31. Control Register Bit Description

Bit	Type	Control Field	Description
0	W	PCTXFIFO_AVAIL	Write the following values: • 1: To indicate that the Channel 0 Posted TXFIFO has a peripheral posted or completion header. And data up to maximum payload size is available. • 0: Clear by hardware.
1	W	NPTXFIFO_AVAIL	Write the following values: • 1: To indicate that the Channel 0 Non-posted TXFIFO has a complete peripheral non-posted header available. • 0: Clear by hardware.
31:2	-	-	Reserved.

7.5.1.3. Error Register

Table 32. Error Register Bit Definition

To clear any of the following register bits, write 1 to that respective bit.

Bit	Type	Error Field
0	R/W1C	Invalid Command Opcode
1	R/W1C	Invalid Cycle Type
		continued...

(9) eSPI host issue command such as PUT_NP, PUT_MEMRD_SHORT, PUT_IORD_SHORT, or PUT_IOWR_SHORT to write data byte into this FIFO. To retrieve data from this FIFO, use Avalon read signal.

Bit	Type	Error Field
2	R/W1C	Command Phase CRC Error
3	R/W1C	Unexpected de-assertion of espi_cs_n
4	R/W1C	Put without Free
5	R/W1C	Get without Available
6	R/W1C	Malformed packed in Peripheral Channel
7	R/W1C	Malformed packed in Virtual Wire Channel
31:8	-	Reserved

7.5.2. eSPI Interface Accessible Registers

7.5.2.1. eSPI Status Register

This status register reflects the status of the eSPI agent IP. You can access this register using the GET_STATUS command through eSPI interface. You can also access this register via Avalon-MM by asserting the read request with Avalon-MM address 00h.

During the response phase, the status field is always returned to the eSPI host. The received command is only decoded after the deassertion of espi_cs_n signal. Therefore, the implementation effect of a queued command is reflected in the status of the subsequent transaction.

The eSPI status register bits are read-only and cleared by espi_reset_n.

Table 33. eSPI Status Register Bit Description

Bit	Status Field	Description
0	PC_FREE	The following values indicates: • 1: Channel 0 Posted FIFO is empty to accept peripheral posted or completion header and data up to maximum payload size. • 0: Channel 0 Posted FIFO has one complete peripheral posted or completion header & data packet.
1	NP_FREE	The following values indicates: • 1: Channel 0 Non-posted FIFO is empty to accept peripheral posted or completion header and data up to maximum payload size. • 0: Channel 0 Non-posted FIFO has one complete peripheral posted or completion header & data packet.
2	VWIRE_FREE	The value is always 1.
3	OOB_FREE	The value is always 1.
4	PC_AVAIL	The following values indicates: • 1: Channel 0 Posted FIFO has a peripheral posted or completion header. And data up to maximum payload size is available to send. • 0: Channel 0 Posted FIFO is empty.
5	NP_AVAIL	The following values indicates:

continued...

Bit	Status Field	Description
		• 1: Channel 0 Non-posted FIFO has a peripheral posted or completion header. And data up to maximum payload size is available to send. • 0: Channel 0 Non-posted FIFO is empty.
6	VWIRE_AVAIL	The following values indicates: • 1: Channel 1 has a tunneled virtual wire available to send. • 0: Channel 1 is empty.
7	OOB_AVAIL	The value is always 0.
8	FLASH_C_FREE	The value is always 1.
9	FLASH_NP_FREE	The value is always 1.
10	Reserved	-
11	Reserved	-
12	FLASH_C_AVAIL	The value is always 0.
13	FLASH_NP_AVAIL	The value is always 0.
14	Reserved	-
15	Reserved	-

7.5.2.2. Capabilities and Configuration Registers

You can access these register using the GET_CONFIGURATION and SET_CONFIGURATION commands. When you configure the registers using the SET_CONFIGURATION command, the new register value takes affect only at the deassertion edge of espi_cs_n.

The capabilities and configuration register bits are reset by espi_reset_n.

Table 34. Capabilities and Configuration Register Map

Offset	Register Name
0x4	Device Identification
0x8	General Capabilities and Configurations
0x10	Channel 0 Capabilities and Configurations
0x20	Channel 1 Capabilities and Configurations
0x30	Channel 2 Capabilities and Configurations
0x40	Channel 3 Capabilities and Configurations

Table 35. Device Identification Register Description

Bit	Access Type	Default Value	Description
31:8	-	-	Reserved.
7:0	R	0x01	Indicates the Version ID that is compliant to the specific eSPI specification revision.

Table 36. General Capabilities and Configurations Register Description

Bit	Access Type	Default Value	Description
31	RW	0	CRC Checking: • 1: CRC checking is enabled • 0: CRC checking is disabled
30:29	-	-	Reserved.
28	RW	0	Alert Mode: • 1: `espi_alert_n` is used to signal the Alert event • 0: `espi_data[1]` is used to signal the Alert event
27:26	RW	2'b00	I/O Mode Select: • 2'b00: Single I/O • 2'b01: Dual I/O • 2'b10: Quad I/O • 2'b11: Reserved
25:24	R	-	Indicates value set for **eSPI Mode of Operation** parameter.
23	-	-	Reserved.
22:20	RW	3'b000	Operating Frequency: • 3'b000: 20MHz • 3'b001: 25MHz • 3'b010: 33MHz • 3'b011: 50MHz • 3'b100: 66MHz
19	-	-	Reserved.
18:16	R	-	Indicates value set for **Frequency of Operation** parameter.
15: 12	RW	4'b0000	The maximum Wait State allowed before responding with an ACCEPT, DEFER, NON_FATAL ERROR or FATAL ERROR response code: • 4'b0000: 16 byte time • 4'b0001: 1 byte time • 4'b0010: 2 byte time • 4'b0011: 3 byte time • ..... • 4'b1111: 15 byte time
11:8	-	-	Reserved.
7:0	R	-	Indicates value set for **Channel Supported** parameter.

Table 37. Channel 0 Capabilities and Configurations Register Description

Bit	Access Type	Default Value	Description
31:15	-	-	Reserved.
14:12	RW	3'b001	Peripheral Channel Address Aligned Maximum Read Request Size: • 3'b000: Reserved • 3'b001: 64 bytes The length of the read request must not cross the naturally aligned address boundary of the corresponding Maximum Read Request Size.
11	-	-	Reserved.

continued...

Bit	Access Type	Default Value	Description
10:8	RW	3'b001	Peripheral Channel Address Aligned Maximum Payload Size Selected: • 3'b000: Reserved • 3'b001: 64 bytes It must never be more than the value stated in the **Peripheral Channel Maximum Payload Size Supported** field.The payload of the transaction must not cross the naturally aligned address boundary of the corresponding Maximum Payload Size.
7	-	-	Reserved.
6:4	R	-	Indicates value set for **Peripheral Channel Maximum Payload Size Supported** parameter.
3:2	-	-	Reserved.
1	R	0	Peripheral Channel Ready: • 1: Channel is ready • 0: Channel is not ready
0	RW	1	Peripheral Channel Enable: • 1: Channel is enabled • 0: Channel is disabled While you clear this bit from 1 to 0, this triggers a reset to the Peripheral Channel.

Table 38. Channel 1 Capabilities and Configurations Register Description

Bit	Access Type	Default Value	Description
31:22	-	-	Reserved.
21:16	RW	0	Operating Maximum Virtual Wire Count - The maximum number of Virtual Wire groups that can be sent in a single Virtual Wire packet. The value configured in this field must never be more than the value stated in **MAX_VW_COUNT**. The default value 0 indicates count of 1. Other legal values: • 6'b000111: 8 count • 6'b001000: 9 count • 6'b001001: 10 count • 6'b001010: 11 count • 6'b001011: 12 count • 6'b001100: 13 count • 6'b001101: 14 count • 6'b001110: 15 count • 6'b001111: 16 count
15:14	-	-	Reserved.
13:8	R	-	Indicates value set for **MAX_VW_COUNT** parameter.
7:2	-	-	Reserved.
1	R	0	Virtual Wire Channel Ready: • 1: Channel is ready • 0: Channel is not ready
0	RW	0	Virtual Wire Channel Enable: • 1: Channel is enabled • 0: Channel is disabled When you clear this bit, it does not reset the Virtual Wire Channel. Therefore, the state of all the Virtual Wires must be maintained.

7.6. Peripheral Channel Avalon Interface Use Model

Each Avalon read command reads back only 8-bit data. Even though the Avalon Memory-Mapped Interface read data is 32-bit wide, but only 8-bits (LSB) are used. This behavior applies to Avalon write command too.

When eSPI host sends PUT_IORD_SHORT or PUT_IOWR_SHORT command, these packets are not stored inside the FIFOs. The packet's data is directly send to `pc_port**_in port or pick from /pc_port**_out` port.

You must use the following format while sending the response packet to PCTXFIFO:

cycletype (SUCCESSFUL_COMPLETION_WITH_DATA/UNSUCCESSFUL_COMPLETION) -> **MSB length** -> **LSB length** -> **DATA** (optional)

After writing a response packet to PCTXFIFO, you must write 1 to Avalon Control Register (0x4h), indicating that the PCTXFIFO has a complete payload available. Once this flag is triggered, the eSPI host is acknowledged (thru espi status information) and fetchs the packet accordingly using the GET_PC command. Each FIFO can only store one packet.

Use Avalon Status Register (0x0h) to verify that the eSPI host sent a complete packet to PCRXFIFO (using PUT_PC/ PUT_MEMWR32_SHORT command) or NPRXFIFO (using PUT_NP/ PUT_MEMRD32_SHORT command).

In the event of the following errors, the FIFO in the peripheral channel is flushed:

- Invalid cycle type
- Invalid command
- CRC mismatch
- Put FIFO without FIFO Free asserted
- Get FIFO without FIFO Available asserted

7.7. Intel eSPI Agent Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2020.07.22	20.2	Corrected the interface signal name callouts.
2018.09.24	18.1	• Corrected signal names (spi_* to espi_*). • Updated on IP clock frequency in Table: *eSPI Agent Core Resource Utilization*. • Updated Avalon interface registers. • Added a new section: *Peripheral Channel Avalon Interface Use Model*,
2018.05.07	18.0	Initial release.

8. eSPI to LPC Bridge Core

The eSPI to LPC Bridge core allows interaction between the eSPI host as the upstream device and LPC agent as the downstream device. This core primarily uses the eSPI agent IP core with some additional blocks to convert the eSPI packets into LPC transactions. This bridge does not use the Avalon interface to access the FIFOs in the peripheral channel. However, the LPC interface signals allow you to connect the bridge with downstream LPC device.

Note: *The eSPI to LPC Bridge Core only supports Intel Arria 10 and Intel MAX 10 devices.*

8.1. Unsupported LPC Features

This IP core does not support the following LPC features:

- DMA Read/Write cycle
- Bus Host Memory Read/Write cycle
- Bus Host IO Read/Write cycle
- LPC clock enable/disable

8.2. IP Parameters

Table 39. Parameter List

You can configure these parameters using the Platform Designer.

Parameter Name	Default Value	Description
eSPI Mode of Operation	Single I/O	Allows you to set I/O configuration. Available options: • Single I/O • Single and Dual I/O • Single and Quad I/O • Single, Dual, and Quad I/O
Frequency of Operation	20MHz	Allows you to set the operational frequency. Available options: • 20MHz • 25MHz • 33MHz • 50MHz • 66MHz
Channel Supported	Peripheral Channel	Allows you to set the channel support. Available option: • Virtual Wire Channel • Peripheral Channel • All Channel
Peripheral Channel Maximum Payload Size Supported	64 bytes	Allows you to set the maximum payload size for peripheral channel. Available options:

continued...

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Parameter Name	Default Value	Description
		• 64 bytes
Peripheral Channel Maximum Read Request Size Supported	64 bytes	Allows you to set the maximum read request size for peripheral channel. Available options: • 64 bytes
Maximum Virtual Wire Count Supported (0-based)	0x07	Allows you to set the maximum virtual wire count. Legal values: • 0x07: 8 count • 0x08: 9 count • 0x09: 10 count • 0x0A: 11 count • 0x0B: 12 count • 0x0C: 13 count • 0x0D: 14 count • 0x0E: 15 count • 0x0F: 16 count
Enable tri-state control ports on LPC data bus and Serial IRQ line(10)	Off	Allow you to export LPC data bus and Serial IRQ signals as tristate conduits instead of bi-directional signals.

8.3. Supported IP Clock Frequency

Table 40. IP Clock Frequency Support

Device Family	eSPI Clock Frequency (MHz)	Minimum IP Clock Frequency (MHz)	Maximum IP Clock Frequency (MHz)
Intel MAX 10	20	60	150
	25	80	150
	33	100	150
	50(11)	150	150
Intel Arria 10	66	200	-

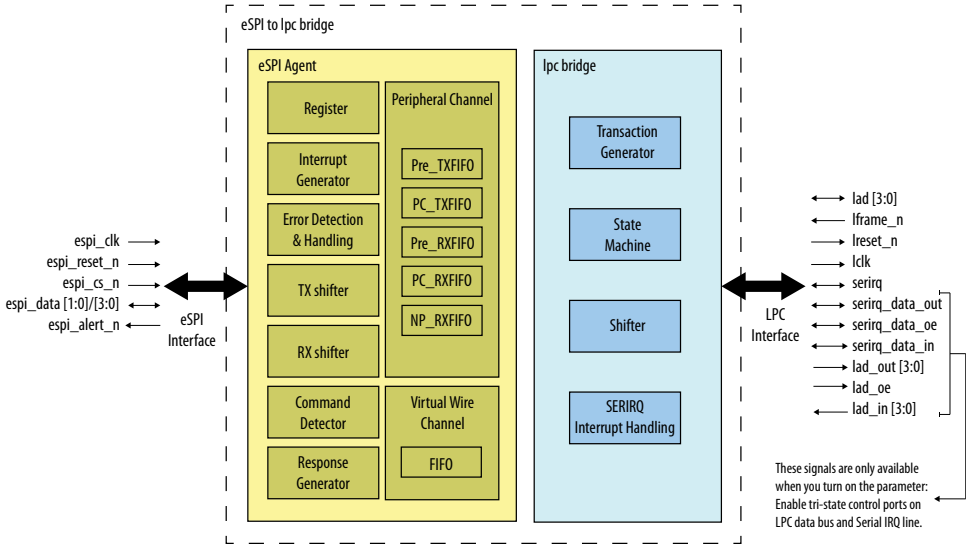
(10) This parameter is available in the Intel Quartus Prime software version 20.1 or later.

(11) You must enable the eSPI host sampling clock at falling edge to increase timing margin for output delay.

8.4. Functional Description

Figure 26. Block Digram

The figure depicts the eSPI agent IP core integrated with the eSPI-to-LPC bridge module.



The `lpc_bridge` block must be used together with the peripheral_channel block. Therefore, you must set the **Channel Supported** parameter bit 0 to **1** while using the `lpc_bridge`.

The SERIRQ interrupt event is sampled and translated into VW index group. Therefore, you must set the **Channel Supported** parameter bit 1 to **1** while using the SERIRQ interrupt.

Note: Enable both channel, peripheral channel and virtual channel, to initialize the LPC bridge internal signal correctly.

8.4.1. FIFO Implementation

All FIFOs are used in peripheral channel to buffer up the incoming transactions. When the eSPI-to-LPC bridge is busy transferring data down the LPC interface, the transactions from the eSPI host are stored using the PC_RXFIFO or NP_RXFIFO until the bridge is idle. The PC_RXFIFO stores posted transactions while the NP_RXFIFO stores non-posted transactions.

The RXFIFOs can store more than one eSPI transaction (command byte, header byte and write data byte) until it is full. When the FIFO depth is less than **MAX_PC_PAYLOAD_SIZE + 1 complete header + 1 command byte** or **1 complete header + 1 command byte for NP_RXFIFO**, the PC_FREE/NP_FREE status register bit is de-asserted.

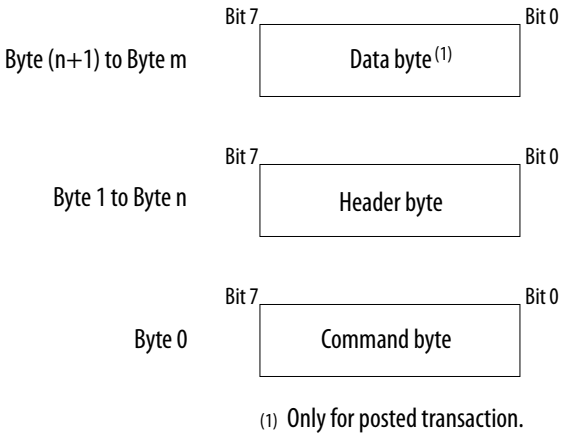
The Pre_RXFIFO stores an incoming eSPI transaction for CRC error checking purposes. CRC error check is performed after pushing the last byte of a eSPI transaction into Pre_RXFIFO. If CRC error is high, then the eSPI transaction is dropped to avoid translation into a LPC transaction. If CRC error is low, the eSPI transaction is pushed into NP_RXFIFO or PC_RXFIFO for translating into a LPC transaction.

The PC_TXFIFO stores response transaction from the downstream LPC devices. The PC_ AVAIL status register bit goes high only when a complete response transaction is stored in the PC_TXFIFO.

The Pre_TXFIFO stores an incoming LPC transaction for error checking purposes. In the event of LPC transaction abort or LPC transaction sync error, the transaction is dropped from the Pre_TXFIFO. The transaction is pushed into PC_TXFIFO provided that there is no error, then to eSPI host.

All the FIFOs are 8-bit wide. The figure below shows the content of the RXFIFO which can store one complete eSPI transaction.

Figure 27. RXFIFO Contents



8.4.2. Transaction Ordering Rule

When the LPC bridge is ready to fetch the transaction from the RXFIFOs, it always pops the transaction from PC_RXFIFO (write transaction) first, then the transaction from NP_RXFIFO (read transaction). This rule prevents any reading of old data.

8.4.3. eSPI Command to LPC Cycle Type Conversion

Table 41. Conversion List

eSPI Command	Supported Sizes (in bytes)	LPC Cycle Type
PUT_NP	1 to 64	Memory Read
PUT_MEMRD32_SHORT_1B	1	
PUT_MEMRD32_SHORT_2B	2	
PUT_MEMRD32_SHORT_4B	4	
PUT_PC	1 to 64	Memory Write
PUT_MEMWR32_SHORT_1B	1	
PUT_MEMWR32_SHORT_2B	2	
PUT_MEMWR32_SHORT_4B	4	

continued...

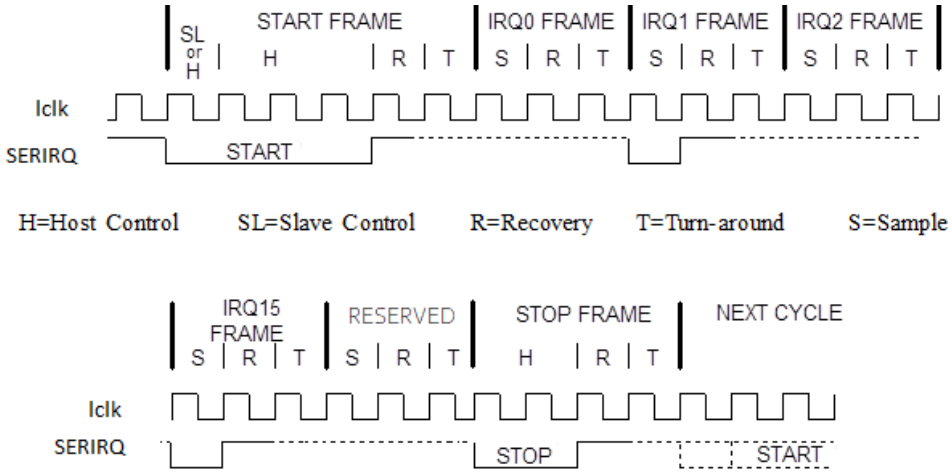
eSPI Command	Supported Sizes (in bytes)	LPC Cycle Type
PUT_IORD_SHORT_1B	1	IO Read
PUT_IORD_SHORT_2B	2	
PUT_IORD_SHORT_4B	4	
PUT_IOWR_SHORT_1B	1	IO Write
PUT_IOWR_SHORT_2B	2	
PUT_IOWR_SHORT_4B	4	

8.4.4. SERIRQ Interrupt Event

The LPC bridge supports 16 lines IRQ/Data serializer by sampling the event of SERIRQ signal from the LPC interface. A SERIRQ cycle transfer consists of three frame types:

- Start Frame: The agent device drives the SERIRQ line low to indicate the start of IRQ transmission.
- IRQ/Data Frames (several): Peripherals transmit the IRQ information. The eSPI bridge host supports 16 IRQ data frames. During the sample phase, the SERIRQ device must drive the SERIRQ low, if and only if, its detected IRQ/Data value is low. And if its detected IRQ/Data value is high, SERIRQ must be left tri-stated. During the recovery phase, the device must drive the SERIRQ high, if and only if, it had driven the SERIRQ low during the sample phase.
- Stop Frame: The Host Controller initiates a Stop frame to terminate SERIRQ activity after all IRQ/Data Frames have completed. A Stop Frame is indicated when the SERIRQ is low for two clock cycles. Stop frame occurs at 53-54 clock cycles past the Start frame.

Figure 28. SERIRQ Event Timing Diagram



The start frame is 4 clock cycles, stop frame is 2 clock cycles, and the IRQ/Data frames clock cycles are stated below:

Table 42. IRQ/Data Frame Clock Period

IRQ/Data Frame	Sampled Signal	Number of Clocks after Start Frame
1	IRQ0	2
2	IRQ1	5
3	SMI#	8
4	IRQ3	11
5	IRQ4	14
6	IRQ5	17
7	IRQ6	20
8	IRQ7	23
9	IRQ8	26
10	IRQ9	29
11	IRQ10	32
12	IRQ11	35
13	IRQ12	38
14	IRQ13	41
15	IRQ14	44
16	IRQ15	47
17	Reserved	-

Once the LPC bridge samples and derives the SERIRQ event, it translates the IRQ/Data Frame into VW index bit as below:

Table 43. SERIRQ IRQ/Data Frame to VW Index Group Mapping

SERIRQ Sampled Signal	VW Index Group	VW Data Bit
SMI#	6h	Bit 2
IRQ0 to IRQ15	00h	Bit 0 - Bit 15

IRQ VW information can be sent to the eSPI host only through GET_VWIRE command during transition period.

Table 44. VW Information via Transition Period

Interrupt Source Type	Interrupt Source Level	Agent to Host IRQ Virtual Wire (Active High)
Active low	0 → 1	De-assertion. IRQ VW (Level='0') sent.
Active low	1 → 0	Assertion. IRQ VW (Level='1') sent.

8.5. Interface Signals

Table 45. Interface Signals

Signal Name	Width	Direction	Description
Clock Interface			
clk	1	Input	Input clock used to clock the IP core.
clk_33	1	Input	33 MHz clock supply to IP core.
Reset Interface			
reset_n	1	Input	Synchronous reset used to reset the IP core.
LPC Interface to LPC Agent Component			
lad[3:0]	4	Input/Output	Multiplexed Command, Address, and Data signal.
lframe_n	1	Output	Indicates start of a new cycle, or termination of broken cycle.
lreset_n	1	Output	Reset signal to LPC agent.
lclk	1	Output	33Mhz clock that drives the LPC agent.
serirq	1	Input/Output	Serialized IRQ interrupt signal.
serirq_data_out(12)	1	Input/Output	Serialized IRQ data signal.
serirq_data_oe(12)	1	Input/Output	Serialized IRQ data output enable signal.
serirq_data_in(12)	1	Input/Output	Serialized IRQ data signal.
lad_out[3:0](12)	4	Output	Multiplexed command, address and data signal
lad_oe(12)	1	Output	Data signal output enable
lad_in[3:0](12)	4	Input	Multiplexed command, address and data signal
Avalon Interface			
avmm_write	1	Input	Avalon write control signal for register access.
avmm_read	1	Input	Avalon read control signal for register access.
avmm_writedata[31:0]	32	Input	Avalon write data bus for register access.
avmm_address[4:0]	5	Input	Avalon address bus for register access.

continued...

(12) This signal is only available when you turn on the parameter: **Enable tri-state control ports on LPC data bus and Serial IRQ line**.

Signal Name	Width	Direction	Description
avmm_readdata[31:0]	32	Output	Avalon read data bus for register access.
eSPI Interface			
espi_clk	1	Input	eSPI serial clock. Frequency range from 20Mhz to 66Mhz.
espi_reset_n	1	Input	eSPI reset. Assertion of this reset signal does not reset the channel's FIFO.
espi_cs_n	1	Input	eSPI chip select.
espi_data[1:0]/[3:0]	2/4	Input/Output	eSPI bidirectional data bus. Data bus lane depending on parameter **IO_MODE**.
espi_alert_n	1	Output	eSPI alert.
Conduit Ports			
slp_s5_n	1	Output	Server platform related signal.
slp_s4_n	1	Output	
slp_s3_n	1	Output	
slp_a_n	1	Output	
slp_lan_n	1	Output	
slp_wlan_n	1	Output	
sus_stat_n	1	Output	
sus_pwrdn_ack	1	Output	
sus_warn_n	1	Output	
pch_to_ec[7:0]	8	Output	
oob_rst_warn	1	Output	
host_rst_warn	1	Output	
smiout_n	1	Output	
nmiout_n	1	Output	
host_c10	1	Output	
pltrst_n	1	Output	
ec_to_pch [7:0]	8	Input	
sus_ack_n	1	Input	
slave_boot_load_done	1	Input	
slave_boot_load_status	1	Input	
oob_rst_ack	1	Input	
wake_n	1	Input	
pme_n	1	Input	

continued...

Signal Name	Width	Direction	Description
sci_n	1	Input	
rcin_n	1	Input	
host_rst_ack	1	Input	
rsmrst_n	1	Input	

8.6. Registers

Table 46. Avalon-MM Interface Accessible Register Map

Offset Address	Register Name	Access Type	Width (bits)
0x0h	Status Register	R	32
0x4h	Error Register	R/W1C	32

8.6.1. Status Register

Table 47. Status Register Bit Definition

Bit	Type	Status Field
0	PC_FREE	• **1**: Channel 0 Posted FIFO is empty to accept peripheral posted or completion header and data up to maximum payload size. • **0**: Channel 0 Posted FIFO has one complete peripheral posted or completion header & data packet.
1	NP_FREE	• **1**: Channel 0 Non-posted FIFO is empty to accept peripheral non-posted or completion header and data up to maximum payload size. • **0**: Channel 0 Non-posted FIFO has one complete peripheral non-posted or completion header & data packet.
2	VWIRE_FREE	Always **1**
3	OOB_FREE	Always **1**
4	PC_AVAIL	• **1**: Channel 0 Posted FIFO has a peripheral posted or completion header and data up to maximum payload size available to send. • **0**: Channel 0 Posted FIFO is empty.
5	NP_AVAIL	Always **0**
6	VWIRE_AVAIL	• **1**: Channel 1 has a tunneled virtual wire available to send (after the logical state of a virtual wire changes, this VW index and data is stored in the Channel 1 TX FIFO) • **0**: Channel 1 is empty.
7	OOB_AVAIL	Always **0**
8	FLASH_C_FREE	Always **1**
9	FLASH_NP_FREE	Always **1**
11:10	RESERVED	-
12	FLASH_C_AVAIL	Always **0**
13	FLASH_NP_AVAIL	Always **0**

8.6.2. Error Register

Table 48. Error Register Bit Definition

To clear any of the following register bits, write 1 to that respective bit.

Bit	Type	Error Field
0	R/W1C	Invalid Command Opcode
1	R/W1C	Invalid Cycle Type
2	R/W1C	Command Phase CRC Error
3	R/W1C	Unexpected de-assertion of `espi_cs_n`
4	R/W1C	Put without Free
5	R/W1C	Get without Available
6	R/W1C	Malformed packed in Peripheral Channel
7	R/W1C	Malformed packed in Virtual Wire Channel
31:8	-	Reserved

8.7. eSPI to LPC Bridge Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2020.09.21	20.2	Clarified the device support for *eSPI to LPC Bridge Core*.
2020.07.22	20.2	Added a new parameter: **Enable tri-state control ports on LPC data bus and Serial IRQ line** and the corresponding interface signal information.
2018.09.24	18.1	Initial release.

9. Ethernet MDIO Core

9.1. Core Overview

The Intel Management Data Input/Output (MDIO) IP core is a two-wire standard management interface that implements a standardized method to access the external Ethernet PHY device management registers for configuration and management purposes. The MDIO IP core is IEEE 802.3 standard compliant.

To access each PHY device, the PHY register address must be written to the register space followed by the transaction data. The PHY register addresses are mapped in the MDIO core's register space and can be accessed by the host processor via the Avalon Memory-Mapped (Avalon-MM) interface. This IP core can also be used with the Intel FPGA 10-Gbps Ethernet MAC and Intel FPGA Triple Speed Ethernet IP Core to realize a fully manageable system.

Related Information

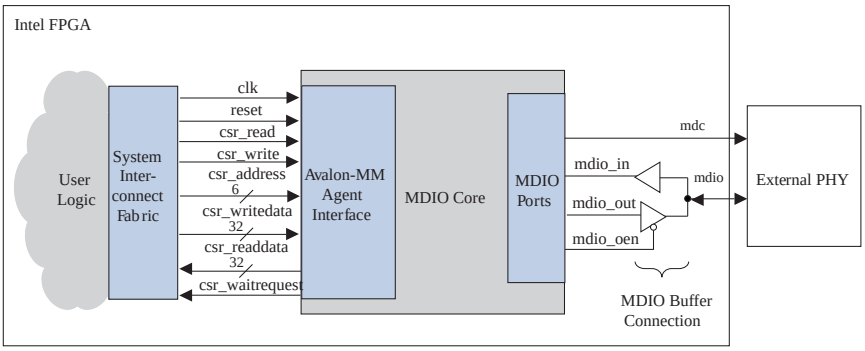
- Intel FPGA Low Latency Ethernet 10G MAC User Guide
- Intel FPGA Triple-Speed Ethernet IP Core User Guide

9.2. Functional Description

The core provides an Avalon Memory-Mapped (Avalon-MM) agent interface that allows Avalon-MM host peripherals (such as a CPU) to communicate with the core and access the external PHY by reading and writing the control and data registers. The system interconnect fabric connects the Avalon-MM host and agent interface while a buffer connects the MDIO interface signals to the external PHY.

For more information about system interconnect fabric for Avalon-MM interfaces, refer to the System Interconnect Fabric for Memory-Mapped Interfaces.

Figure 29. MDIO Core Block Diagram



Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

9.2.1. MDIO Frame Format (Clause 45)

The MDIO core communicates with the external PHY device using frames. A complete frame is 64 bits long and consists of 32-bit preamble, 14-bit command, 2-bit bus direction change, and 16-bit data. Each bit is transferred on the rising edge of the management data clock (MDC). The PHY management interface supports the standard MDIO specification (IEEE802.3 Ethernet Standard Clause 45).

Figure 30. MDIO Frame Format (Clause 45)

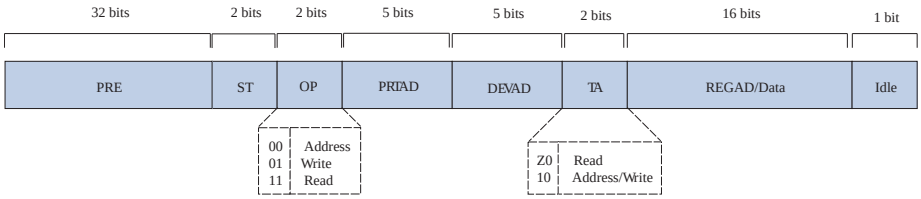


Table 49. MDIO Frame Field Descriptions—Clause 45

Field Name	Description
PRE	Preamble. 32 bits of logical 1 sent prior to every transaction.
ST	The start of frame for indirect access cycles is indicated by the <00> pattern. This pattern assures a transition from the default one and identifies the frame as an indirect access.
OP	The operation code field indicates the following transaction types: 00 indicates that the frame payload contains the address of the register to access. 01 indicates that the frame payload contains data to be written to the register whose address was provided in the previous address frame. 11 indicates that the frame is a read operation. The post-read-increment-address operation <10> is not supported in this frame.
PRTAD	The port address (PRTAD) is 5 bits, allowing 32 unique port addresses. Transmission is MSB to LSB. A station management entity (STA)[13] must have a prior knowledge of the appropriate port address for each port to which it is attached, whether connected to a single port or to multiple ports.
DEVAD	The device address (DEVAD) is 5 bits, allowing 32 unique MDIO manageable devices (MMDs)[14] per port. Transmission is MSB to LSB.
TA	The turnaround time is a 2-bit time spacing between the device address field and the data field of a management frame to avoid contention during a read transaction. For a read transaction, both the STA and the MMD remain in a high-impedance state (Z) for the first bit time of the turnaround. The MMD drives a 0 during the second bit time of the turnaround of a read or postread-increment-address transaction.

continued...

(13) The device driving the MDIO bus is identified as the Station Management Entity (STA).

(14) The target devices managed by the MDC are referred as MDIO Manageable Devices (MMDs).

Field Name	Description
	For a write or address transaction, the STA drives a 1 for the first bit time of the turnaround and a 0 for the second bit time of the turnaround.
REGAD/ Data	The register address (REGAD) or data field is 16 bits. For an address cycle, it contains the address of the register to be accessed on the next cycle. For the data cycle of a write frame, the field contains the data to be written to the register. For a read frame, the field contains the contents of the register. The first bit transmitted and received is bit 15.
Idle	The idle condition on MDIO is a high-impedance state. All tri-state drivers are disabled and the MMDs pullup resistor pulls the MDIO line to a one.

9.2.2. MDIO Clock Generation

The MDIO core's MDC is generated from the Avalon-MM interface clock signal, `clk`. The `MDC_DIVISOR` parameter specifies the division parameter. For more information about the parameter, refer to the **Parameter** section.

The division factor must be defined such that the MDC frequency does not exceed 2.5 MHz.

9.2.3. Interfaces

The MDIO core consists of a single Avalon-MM agent interface. The agent interface performs Avalon-MM read and write transfers initiated by an Avalon-MM host in the client application logic. The Avalon-MM agent uses the `waitrequest` signal to implement backpressure on the Avalon-MM host for any read or write operation which has yet to be completed.

For more information about Avalon-MM interfaces, refer to the Avalon Interface Specifications.

9.2.4. Operation

The MDIO core has bidirectional external signals to transfer data between the external PHY and the core.

9.2.4.1. Write Operation

Follow the steps below to perform a write operation.

1. Issue a write to the device register at address offset `0x21` to configure the device, port, and register addresses of the PHY.
2. Issue a write to the `MDIO_ACCESS` register at address offset `0x20` to generate an MDIO frame and write the data to the selected PHY device's register.

9.2.4.2. Read Operation

Follow the steps below to perform a read operation.

1. Issue a write to the device register at address offset `0x21` to configure the device, port, and register addresses of the PHY.
2. Issue a read to the `MDIO_ACCESS` register at address offset `0x20` to read the selected PHY device's register.

9.3. Parameter

Table 50. Configurable Parameter

Parameter	Legal Values	Default Value	Description
MDC_DIVISOR	8-64	32	The host clock divisor provides the division factor for the clock on the Avalon-MM interface to generate the preferred MDIO clock (MDC). The division factor must be defined such that the MDC frequency does not exceed 2.5 MHz. Formula: For example, if the Avalon-MM interface clock source is 100 MHz and the desired MDC frequency is 2.5 MHz, specify a value of 40 for the MDC_DIVISOR.

9.4. Configuration Registers

An Avalon-MM host peripheral, such as a CPU, controls and communicates with the MDIO core via 32-bit registers, shown in the **Register Map** table.

Table 51. Register Map

Address Offset	Bit(s)	Name	Access Mode	Description
0x00-0x1F	31:0	Reserved	RW	Reserved for future use.
0x20 (1)	31:0	MDIO_ACCESS	RW	Performs a read or write of 32-bit data to the external PHY device. The addresses of the external PHY device's register, device, and port are specified in address offset 0x21.
0x21 (2)	4:0	MDIO_DEVAD	RW	Contains the device address of the PHY.
	7:5	Reserved	RW	Unused.
	12:8	MDIO_PRTAD	RW	Contains the port address of the PHY.
	15:13	Reserved	RW	Unused.
	31:16	MDIO_REGAD	RW	Contains the register address of the PHY.

Note :
1. The byte address for this register is 0x80.
2. The byte address for this register is 0x84.

9.5. Interface Signals

Table 52. Ethernet MDIO Core Signal Description

Signal	Width	Direction	Description
Clock			
clk	1	Input	Avalon-MM interface clock signal.
Reset			
reset	1	Input	Asynchronous reset for Ethernet MDIO core.
Avalon-MM Agent Interface for CSR			
csr_write	1	Input	Avalon-MM write control signal.

continued...

Signal	Width	Direction	Description
csr_read	1	Input	Avalon-MM read control signal.
csr_address	6	Input	Avalon-MM address bus.
csr_writedata	32	Input	Avalon-MM write data bus.
csr_readdata	32	Output	Avalon-MM read data bus.
csr_waitrequest	1	Output	Avalon-MM wait-request control signal..
MDIO Interface			
mdio_in	1	Input	Management data input to FPGA bidirectional I/O buffer.
mdio_out	1	Output	Management data output from FPGA bidirectional I/O buffer.
mdio_oen	1	Output	An active low management data output enable signal to FPGA bidirectional I/O buffer. It is used to enable mdio_in or mdio_out.
mdc	1	Output	Management data clock. This clock is generated from Avalon-MM interface clock signal, clk. Use the MDC_DIVISOR parameter to specify the division factor such that the frequency of this clock does not exceed 2.5MHz.

9.6. Ethernet MDIO Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0	Added the section *Interface Signals*.

Date	Version	Changes
July 2014	2014.07.24	Removed mention of SOPC Builder, updated to Platform Designer
December 2010	v10.1.0	Revised the register map address offset.
July 2010	v10.0.0	Initial release.

10. Intel FPGA 16550 Compatible UART Core

10.1. Core Overview

The Intel FPGA 16550 UART (Universal Asynchronous Receiver/Transmitter) soft IP core with Avalon interface is designed to be register space compatible with the de-facto standard 16550 found in the PC industry. The core provides RS-232 Signaling interface, False start detection, Modem control signal and registers, Receiver error detection and Break character generation/detection. The core also has an Avalon Memory-Mapped (Avalon-MM) agent interface that allows Avalon-MM host peripherals (such as a Nios II processor) to communicate with the core simply by reading and writing control and data registers.

The 16550 UART supports all memory types depending on the device family.

Note: You must acquire license to use this core..

Table 53. Product Information

Core	Product ID
Intel FPGA 16550 UART (Universal Asynchronous Receiver/ Transmitter) soft IP core	6af7 010c

Note: This IP core supports Intel eASIC N5X devices.

10.2. Feature Description

The 16550 Soft-UART has the following features:

- RS-232 signaling interface
- Avalon-MM agent
- Single clock
- False start detection
- Modem control signal and registers
- Receiver error detection
- Break character generation/detection
- Supports full duplex mode by default

Table 54. UART Features and Configurability

Features	Run Time Configurable	Generate Time Configurable
FIFO/FIFO-less mode	Yes	Yes
FIFO Depth	-	Yes

continued...

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Features	Run Time Configurable	Generate Time Configurable
5-9 bit character length	Yes	-
1, 1.5, 2 character stop bit	Yes	-
Parity enable	Yes	-
Even/Odd parity	Yes	-
Baud rate selection	Yes	-
Memory Block Type	-	Yes
Priority based interrupt with configurable enable	Yes	-
Hardware Auto Flow Control (cts_n/ rts_n signals)	Yes	Yes
DMA Extra (configurable support for extra DMA sideband signal)	Yes	Yes
Stick parity/Force parity	Yes	-

Note: When a feature is both Generate time and Run time configurable, the feature must be enabled during Generate time before Run time configuration can be used. Therefore, turning ON a feature during Generate time is a prerequisite to enabling/disabling it during run time.

10.2.1. Unsupported Features

Unsupported Features vs PC16550D:

- Separate receive clock
- Baud clock reference output

10.2.2. Interface

The Soft UART will have the following signal interface, exposed using `_hw.tcl` through Platform Designer software.

Table 55. Clock and Reset Signal Interface

Pin Name	Direction	Description
`clk`	Input	Avalon clock sink
`rst_n`	Input	Avalon reset sink Asynchronous assert, Synchronous deassert active low reset. Interconnect fabric expected to perform synchronization – UART and interconnect is expected to be placed in the same reset domain to simplify system design

Table 56. Avalon-MM Agent

Pin Name	Width	Direction	Description
`addr`	9	Input	Avalon-MM Address bus

continued...

Pin Name	Width	Direction	Description
			Highest addressable byte address is 0x118 so a 9-bit width is required
read		Input	Avalon-MM Read indication
readdata	32	Output	Avalon-MM Read Data Response from the agent
write		Input	Avalon-MM Write indication
writedata	32	Input	Avalon-MM Write Data

Table 57. Interrupt Interface

Pin Name	Direction	Description
intr	Output	Interrupt signal

Table 58. Flow Control

Pin Name	Direction	Description
sin	Input	Serial Input from external link.
sout	Output	Serial Output to external link.
sout_oe	Output	Output enable for Serial Output to external link. sout_oe signal will be high when the UART is transmitting and low when the UART is IDLE.

Table 59. Modem Control and Status

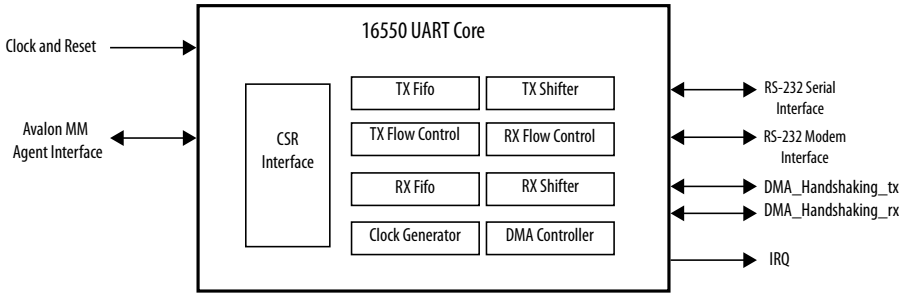
Pin Name	Direction	Description
cts_n	Input	Clear to Send
rts_n	Output	Request to Send
dsr_n	Input	Data Set Ready
dcd_n	Input	Data Carrier Detect
ri_n	Input	Ring Indicator
dtr_n	Output	Data Terminal Ready
out1_n	Output	User Designated Output1
out2_n	Output	User Designated Output2

Table 60. DMA Sideband Signals

Pin Name	Direction	Description
dma_tx_ack_n	Input	TX DMA acknowledge
dma_rx_ack_n	Input	RX DMA acknowledge
dma_tx_req_n	Output	TX DMA request
dma_rx_req_n	Output	RX DMA request
dma_tx_single_n	Output	TX DMA single request
dma_rx_single_n	Output	RX DMA single request

10.2.3. General Architecture

Figure 31. Soft-UART High Level Architecture



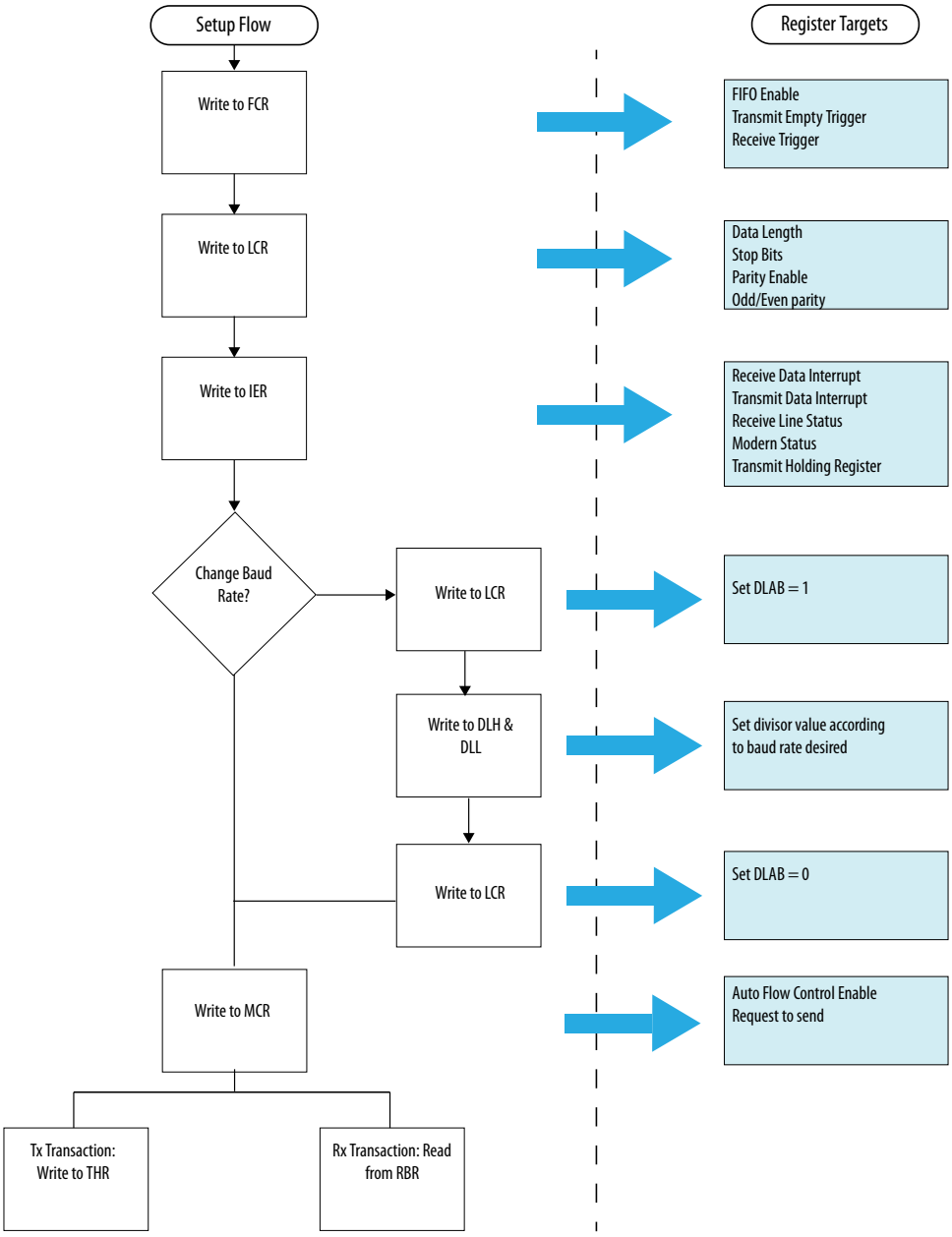
The figure above shows the high level architecture of the UART IP. Both Transmit and Receive logic have their own dedicated control & data path. An interrupt block and clock generator block is also present to service both transmit and receive logic.

10.2.4. 16550 UART General Programming Flow Chart

The 16550 UART general programming flow chart is the recommended flow for setting up the UART for error free operation.

Note: You are free to change this flow to fit your own usage model but the changes might cause undefined results.

Figure 32. 16550 UART Configuration Flow



For more information on the register descriptions used in the flow chart, refer to the "Address Map and Register Descriptions" section.

Related Information

Address Map and Register Descriptions on page 114

10.2.5. Configuration Parameters

The table below shows all the parameters that can be used to configure the UART. (_hw.tcl) is the mechanism used to enforce and validate correct parameter settings.

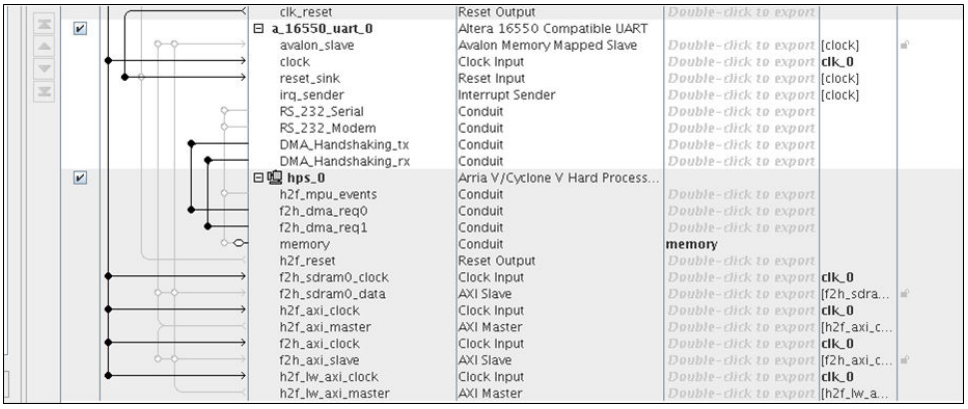
Table 61. Configuration Parameters

Parameter Name	Description	Default
MEM_BLOCK_TYPE	Set memory block type of FIFO. Available memory block depend on device family used. FIFO_MODE must be 1	AUTO
FIFO_MODE	1 = FIFO mode enabled 0 = FIFO mode disabled	1
FIFO_DEPTH	Set depth of FIFO Values limited to 32, 64, 128, and 256 FIFO_MODE must be 1	128
FIFO_HWFC	1 = Enabled hardware flow control 0 = Disabled hardware flow control Mutually exclusive with FIFO_SWFC FIFO_MODE must be 1	1
DMA_EXTRA	1 = Additional DMA interface enabled 0 = Additional DMA interface disabled	0

10.2.6. DMA Support

The DMA interface (DMA_EXTRA) is disabled by default. It must be enabled in the IP to have the additional DMA_Handshaking_tx and DMA_Handshaking_rx interfaces. DMA support is only available when used with the HPS DMA controller. The HPS DMA controller has the required handshake signals to control DMA data transfers with the IP through the DMA_Handshaking_tx and DMA_Handshaking_rx interfaces. The DMA handshaking interfaces are connected to the HPS through the f2h DMA request lines.

Figure 33. Intel FPGA 16550 UART's DMA Handshaking Interfaces Connection to Arria V/Cyclone V HPS in Platform Designer



For more information about the HPS DMA Controller handshake signals, refer to the DMA Controller chapter in the *Cyclone V Device Handbook, Volume* 3.

Related Information

DMA Controller Core on page 322

10.2.7. FPGA Resource Usage

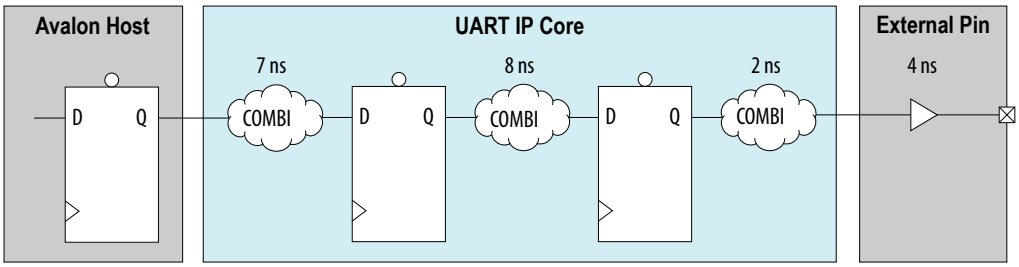
In order to optimize resource usage, in terms of register counts, the UART IP design specifically targets MLABs to be used as FIFO storage element. The following table lists the FPGA resources required for one UART with 128 Byte Tx and Rx FIFO.

Table 62. UART Resource Usage

Resource	Number
ALMS needed	362
Total LABs	54
Combinational ALUT usage for logic	436
Combinational ALUT usage for route-throughs	17
Dedicated logic registers • Design implementation registers (294) • Routing optimization registers (17)	311
Global Signals	2
M10k blocks	0
Total MLAB memory bits	2432

10.2.8. Timing and Fmax

Figure 34. Maximum Delays on UART



The diagram above shows worst case combinatorial delays throughout the UART IP Core. These estimates are provided by Timing Analyzer under the following condition:

- Device Family: Series V and above
- Avalon Host connected to Avalon Agent port of the UART with outputs from the Avalon Host registered
- RS-232 Serial Interface is exported to FPGA Pin
- Clocks for entire system set at 125 MHz

Based on the conditions above the UART IP has an Fmax value of 125 MHz, with the worst delay being internal register-to-register paths.

The UART has combinatorial logic on both the Input and Output side, with system level implications on the Input side.

The Input side combinatorial logic (with 7ns delay) goes through the Avalon address decode logic, to the Read data output registers. It is therefore recommended that Hosts connected to the UART IP register their output signals.

The Output side combinatorial logic (with 2ns delay) goes through the RS-232 Serial Output. There should not be any concern on the output side delays though – as it is not a single cycle path. Using the highest clock divider value of 1, the serial output only toggles once every 16 clocks. This naturally gives a 16 clock multi-cycle path on the output side. Furthermore, divider of 1 is an unlikely system, if the UART is clocked at 125 MHz, the resulting baud rate would be 7.81 Mbps.

10.2.9. Avalon-MM Agent

The Avalon-MM Agent has the following configuration:

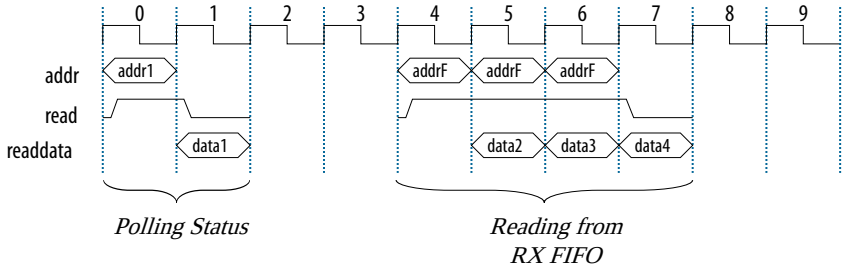
Table 63. Avalon-MM Agent Configuration

Feature	Configuration
Bus Width	32-bit
Burst Support	No burst support. Interconnect is expected to handle burst conversion
Fixed read and write wait time	0 cycles
Fixed read latency	1 cycle
Fixed write latency	0 cycles
Lock support	No

Note: The Avalon-MM interface is intended to be a thin, low latency layer on top of the registers.

10.2.9.1. Read behavior

Figure 35. Reading UART over Avalon-MM

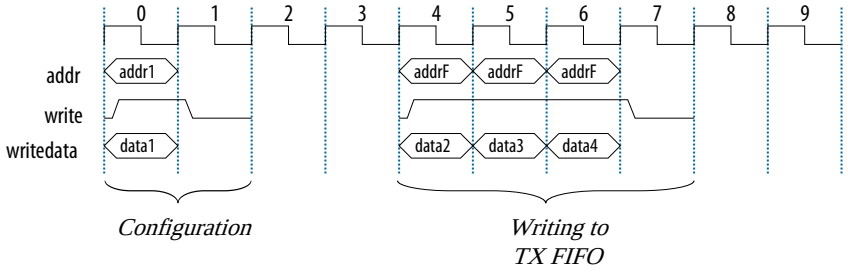


Reads are expected to have 2 types of behavior:

- When status registers are being polled, Reads are expected to be done in singles
- When data needs to be read out from the Rx FIFO, Reads are expected as back-to-back cycles to the same address (these back-to-back reads are likely generated as Fixed Bursts in AXI – but translated into INCR with length of 1 by FPGA interconnect)

10.2.9.2. Write behavior

Figure 36. Writing to UART over Avalon-MM



Writes to the UART are expected as singles during setup phase of any transaction and as back-to-back writes to the same address when the Tx FIFO needs to be filled.

10.2.10. Over-run/Under-run Conditions

Consistent with UART implementation in PC16550D, the soft UART will not implement over-run or under-run prevention on the Avalon-MM interface.

Preventing over-runs and under-runs on the Avalon-MM interface by back-pressuring a pending transaction may cause more harm than good as the interconnect can be held up by the far slower UART.

10.2.10.1. Overrun

On receive path, interrupts can be triggered (when enabled) when overrun occurs. In FIFO-less mode, overrun happens when an existing character in the receive buffer is overwritten by a new character before it can be read. In FIFO mode, overrun happens when the FIFO is full and a complete character arrives at the receive buffer.

On transmit path, software driver is expected to know the Tx FIFO depth and not overrun the UART.

10.2.10.2. Receive Overrun Behavior

When receive overrun does happen, the Soft-UART handles it differently depending on FIFO mode. With FIFO enabled, the newly receive data at the shift register is lost. With FIFO disabled, the newly received data from the shift register is written onto the Receive Buffer. The existing data in the Receive Buffer is overwritten. This is consistent with published PC16550D UART behavior.

10.2.10.3. Transmit Overrun Behavior

When the host CPU forcefully triggers a transmit Overrun, the Soft-UART handles it differently depending on FIFO mode. With FIFO enabled, the newly written data is lost. With FIFO disabled, the newly written data will overwrite the existing data in the Transmit Holding Register.

10.2.10.4. Underrun

No mechanisms exist to detect or prevent under-run.

On transmit path, an interrupts (when enabled) can be generated when the transmit holding register is empty or when the transmit FIFO is below a programmed level.

On receive path, the software driver is expected to read from the UART receive buffer (FIFO-less) or the Rx FIFO based on interrupts (when enabled) or status registers indicating presence of receive data (Data Ready bit, LSR[0]). If reads to Receive Buffer Register is triggered with data ready register being zero, undefined read data is returned.

10.2.11. Hardware Auto Flow-Control

Hardware based auto flow-control uses 2 signals (`cts_n` & `rts_n`) from the Modem Control/Status group. With Hardware auto flow-control disabled, these signals will directly drive the Modem Status register (`cts_n`) or be driven by the Modem Control register (`rts_n`).

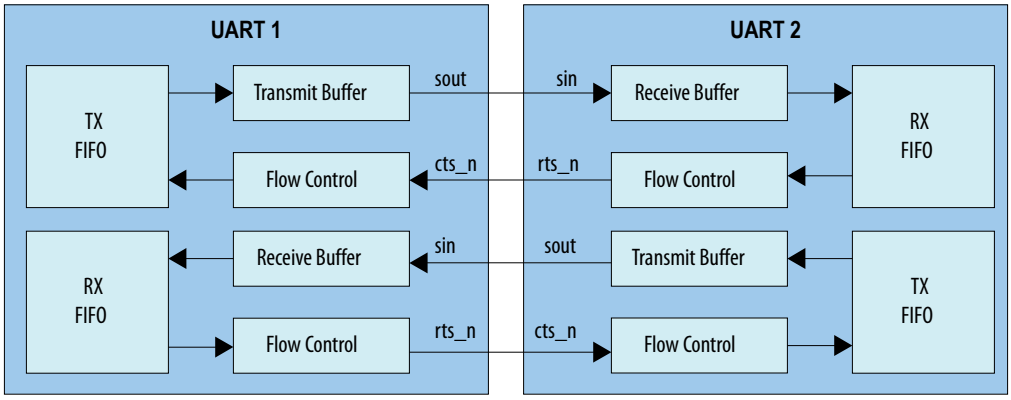
With auto flow-control enabled, these signals perform flow-control duty with another UART at the other end.

The `cts_n` input is, when active (low state), will allow the Tx FIFO to send data to the transmit buffer. When `cts_n` is inactive (high state), the Tx FIFO stops sending data to the transmit buffer. `cts_n` is expected to be connected to the `rts_n` output of the other UART.

The `rts_n` output will go active (low state), when the Rx FIFO is empty, signaling to the opposite UART that it is ready for data. The `rts_n` output goes inactive (high state) when the Rx FIFO level is reached, signaling to the opposite UART that the FIFO is about to go full and it should stop transmitting.

Due to the delays within the UART logic, one additional character may be transmitted after `cts_n` is sampled active low. For the same reason, the Rx FIFO will accommodate up to 1 additional character after asserting `rts_n` (this is allowed because Rx FIFO trigger level is at worst, two entries from being truly full). Both are observed to prevent overflow/underflow between UARTs.

Figure 37. Hardware Auto Flow-Control Between two UARTs



10.2.12. Clock and Baud Rate Selection

The Soft-UART supports only one clock. The same clock is used on the Avalon-MM interface and will be used to generate the baud clock that drives the serial UART interface.

The baud rate on the serial UART interface is set using the following equation:

Baud Rate = Clock/(16 x Divisor)

The table below shows how several typical baud rates can be achieved by programming the divisor values in Divisor Latch High and Divisor Latch Low register.

Table 64. UART Clock Frequency, Divider value and Baud Rate Relationship

	18.432 MHz		24 MHz		50 MHz	
Baud Rate	Divisor for 16x clock	% Error (baud)	Divisor for 16x clock	% Error (baud)	Divisor for 16x clock	% Error (baud)
9,600	120	0.00%	156	0.16%	326	-0.15%
38,400	30	0.00%	39	0.16%	81	0.47%
115,200	10	0.00%	13	0.16%	27	0.47%

10.3. Software Programming Model

10.3.1. Overview

The following describes the programming model for the Intel FPGA compatible 16550 Soft-UART.

10.3.2. Supported Features

For the following features, the 16550 Soft-UART HAL driver can be configurable in run time or generate time. For run-time configuration, users can use “altera_16550_uart_config” API . Generate time is during Platform Designer generation, that is to say once FIFO Depth is selected the depth for the FIFO can’t be change anymore.

Table 65. Supported Features

Features	Run Time	Generate Time
FIFO/ FIFO-less mode	Yes	Yes
FIFO Depth	-	Yes
Programmable Tx/Rx FIFO Threshold	Yes	-
5-9 bit character length	Yes	-
1, 1.5, 2 character stop bit	Yes	-
Parity enable	Yes	-
Even/Odd parity	Yes	-

continued...

Features	Run Time	Generate Time
Stick parity	Yes	-
Baud rate selection	Yes	-
Priority based interrupt with configurable enable	Yes	-
Hardware Auto Flow Control	Yes	Yes

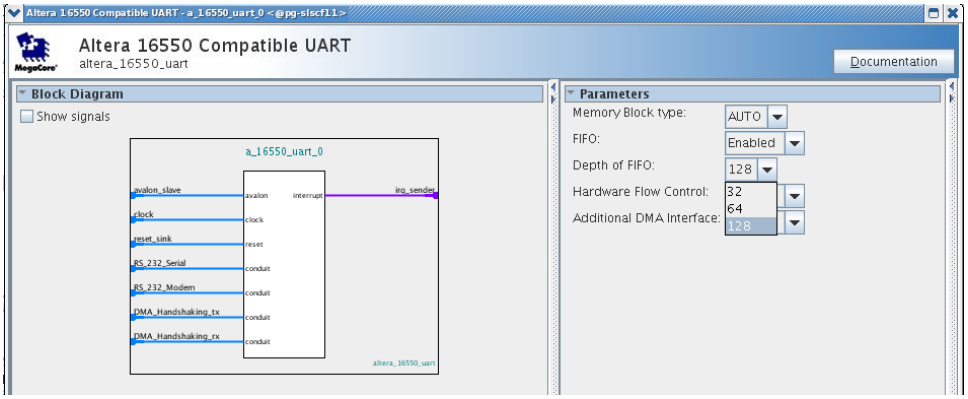
10.3.3. Unsupported Features

The 16550 UART driver does not support Software flow control.

10.3.4. Configuration

The figure below shows the Platform Designer setup on the 16550 Soft-UART's FIFO Depth

Figure 38. Platform Designer Setting to Configure FIFO Depth



10.3.5. 16550 UART API

10.3.5.1. Public APIs

Table 66. altera_16550_uart_open

Prototype:	altera_16550_uart_state* altera_16550_uart_open (const char *name);
Include:	<altera_16550_uart.h>
Parameters:	name—the 16550 UART device name to open.
Returns:	Pointer to 16550 UART or NULL if fail to open
Description	Open 16550 UART device.

Table 67. altera_16550_uart_close

Prototype:	int altera_16550_uart_close(altera_16550_uart_state* sp, int flags);
Include:	<altera_16550_uart.h>
Parameters:	sp—the 16550 UART device name to close. flags—for indicating blocking/non-blocking access for single/ multi threaded.
Returns:	None
Description:	Closes 16550 UART device.

Table 68. altera_16550_uart_read

Prototype:	int altera_16550_uart_read(altera_16550_uart_state* sp, wchar_t* ptr, int len, int flags);
Include:	<altera_16550_uart.h>
Parameters:	sp - The UART device ptr – destination address len – maximum length of the data flags – for indicating blocking/non-blocking access for single/ multi threaded
Returns:	Number of bytes read
Description:	Read data to the UART receiver buffer. UART required to be in a known settings prior executing this function

Table 69. altera_16550_uart_write

Prototype:	int altera_16550_uart_write(altera_16550_uart_state* sp, const wchar_t* ptr, int len, int flags);
Include:	<altera_16550_uart.h>
Parameters:	sp - The UART device ptr – source address len – maximum length of the data flags – for indicating blocking/non-blocking access for single/ multi threaded
Returns:	Number of bytes written
Description:	Writes data to the UART transmitter buffer. UART required to be in a known settings prior executing this function

Table 70. alt_16550_uart_config

Prototype:	alt_u32 alt_16550_uart_config(altera_16550_uart_state* sp, UartConfig *setting);
Include:	<altera_16550_uart.h>
Parameters:	sp - The UART device setting – UART configuration structure to configure UART (refer to UART device structure)
Returns:	Return 0 for success otherwise fail
Description:	Configure UART per user input before initiating read or write

10.3.5.2. Private APIs

Table 71. altera_16550_uart_irq

Prototype:	static void altera_16550_uart_irq (void* context)
Include:	<altera_16550_uart.h>
Parameters:	context – device of the UART
Returns:	none
Description:	Interrupt handler to process UART interrupts to process receiver/transmit interrupts.

Table 72. altera_16550_uart_rxirq

Prototype:	void altera_16550_uart_rxirq(altera_16550_uart_state* sp)
Include:	<altera_16550_uart.h>
Parameters:	sp – UART device
Returns:	none
Description:	Process a receive interrupt. It transfers the incoming character into the receiver circular buffer, and sets the appropriate flags to indicate that there is data ready to be processed.

Table 73. altera_16550_uart_txirq

Prototype:	void altera_16550_uart_txirq(altera_16550_uart_state* sp)
Include:	<altera_16550_uart.h>
Parameters:	sp – UART device
Returns:	none
Description:	Process a transmit interrupt. It transfers data from the transmit buffer to the device, and sets the appropriate flags to indicate that there is data ready to be processed.

10.3.5.3. UART Device Structure

Example 1. UART Device Structure 1

```
typedef enum stopbit { STOPB_1 = 0,STOPB_2 } StopBit;
typedef enum paritybit { ODD_PARITY = 0, EVEN_PARITY, MARK_PARITY,
SPACE_PARITY, NO_PARITY } ParityBit;
typedef enum databit { CS_5 = 0, CS_6, CS_7, CS_8, CS_9 = 256} DataBit;
typedef enum baud
{
BR9600 = B9600,
BR19200 = B19200,
BR38400 = B38400,
BR57600 = B57600,
BR115200 = B115200
} Baud;
typedef enum rx_fifo_level_e { RXONECHAR = 0, RXQUARTER, RXHALF, RXFULL }
Rx_FifoLvl;
typedef enum tx_fifo_level_e { TXEMPTY = 0, TXTWOCHAR, TXQUARTER, TXHALF }
Tx_FifoLvl;
typedef struct uart_config_s
{
StopBit stop_bit;
```

```
ParityBit parity_bit;
DataBit data_bit;
Baud baudrate;
alt_u32 fifo_mode;
Rx_FifoLvl rx_fifo_level;
Tx_FifoLvl tx_fifo_level;
alt_u32 hwfc;
} UartConfig;
```

Example 2. UART Device Structure 2

```
typedef struct altera_16550_uart_state_s
{
alt_dev dev;
void* base; /* The base address of the device */
alt_u32 clock;
alt_u32 hwfifomode;
alt_u32 ctrl; /* Shadow value of the LSR register */
volatile alt_u32 rx_start; /* Start of the pending receive data */
volatile alt_u32 rx_end; /* End of the pending receive data */
volatile alt_u32 tx_start; /* Start of the pending transmit data */
volatile alt_u32 tx_end; /* End of the pending transmit data */
alt_u32 freq; /* Current clock freq rate */
UartConfig config; /* Uart setting */
#ifdef ALTERA_16550_UART_USE_IOCTL
struct termios termios;
#endif
alt_u32 flags; /* Configuration flags */
ALT_FLAG_GRP (events) /* Event flags used for
* foreground/background in mult-threaded
* mode */
ALT_SEM (read_lock) /* Semaphore used to control access to the
* read buffer in multi-threaded mode */
ALT_SEM (write_lock) /* Semaphore used to control access to the
* write buffer in multi-threaded mode */
volatile wchar_t rx_buf[ALT_16550_UART_BUF_LEN]; /* The receive buffer */
volatile wchar_t tx_buf[ALT_16550_UART_BUF_LEN]; /* The transmit buffer */
line_status_reg line_status; /* line register status for the current read byte
data of RBR or data at the top of FIFO*/
alt_u8 error_ignore; /* received data will be discarded
for the current read byte data of RBR or data at the top of FIFO if pe, fe and
bi errors detected after error_ignore is set to '0' */
} altera_16550_uart_state;
```

10.3.6. Driver Examples

Below is a simple test program to verify that the Intel FPGA 16550 UART driver support is functional.

The test reads, validates, and writes a modified baud rate, data bits, stop bits, parity bits to the UART before attempting to write a character stream to it from UART0 to UART1 and vice verse (ping pong test). This also tests the FIFO and FIFO-less mode to ensure the IP is functioning for FIFO.

Prerequisites needed before running test:

- An instance of UART named "a_16550_uart_0" and another instance UART named "a_16550_uart_1".
- Both UARTs need to be connected in loopback in Intel Quartus Prime software.

Additional coverage:

- Non-blocking UART support
- UART HAL driver
- HAL open/write support

The test will print "ALL TESTS PASS" from the UART to indicate success.

Example 3. Verifying Intel FPGA 16550 UART Driver Support functionality

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/termios.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <sys/time.h>
#include <time.h>
#include "system.h"
#include "altera_16550_uart.h"
#include "altera_16550_uart_regs.h"
#include <wchar.h>

#define ERROR -1
#define SUCCESS 0
#define MOCK_UART
#define BUFSIZE 512
wchar_t TXMessage[BUFSIZE] = L"Hello World";
wchar_t RXMessage[BUFSIZE] = L"";

int UARTDefaultConfig(UartConfig *Config)
{
  Config->stop_bit     = STOPB_1;
  Config->parity_bit   = NO_PARITY;
  Config->data_bit     = CS_8;
  Config->baudrate     = BR115200;
  Config->fifo_mode    = 0;
  Config->hwfc         = 0;
  Config->rx_fifo_level= RXFULL;
  Config->tx_fifo_level= TXEMPTY;
  return 0;
}

int UARTBaudRateTest()
{
  UartConfig *UART0_Config = malloc(1*sizeof(UartConfig));
  UartConfig *UART1_Config = malloc(1*sizeof(UartConfig));
```

```
  int i=0, j=0, direction=0, Match=0;
  const int nBaud = 5;
  int BaudRateCoverage[]= {BR9600, BR19200, BR38400, BR57600, BR115200};
  altera_16550_uart_state* uart_0;
  altera_16550_uart_state* uart_1;

  printf("============ UART Baud Rate Test Starts Here ===============\n");
  uart_0 = altera_16550_uart_open ("/dev/a_16550_uart_0");
  uart_1 = altera_16550_uart_open ("/dev/a_16550_uart_1");

  for (direction=0; direction<2; direction++)
  {
      for (i=0; i<nBaud; i++)
          {
             UARTDefaultConfig(UART0_Config);
            UARTDefaultConfig(UART1_Config);
            UART0_Config->baudrate=BaudRateCoverage[i];
            UART1_Config->baudrate=BaudRateCoverage[i];
            printf("Testing Baud Rate: %d\n", UART0_Config->baudrate);
            if(ERROR == alt_16550_uart_config (uart_0, UART0_Config)) return
ERROR;
            if(ERROR == alt_16550_uart_config (uart_1, UART1_Config)) return
ERROR;

            switch(direction)
            {
             case 0:
                printf("Ping Pong Baud Rate Test: UART#0 to UART#1\n");
                for(j=0; j<wcslen(TXMessage); j++)
                {
                    altera_16550_uart_write(uart_0, &TXMessage[j], 1, 0);
                    usleep(1000);
                    if(ERROR== altera_16550_uart_read(uart_1,  RXMessage, 1,
0)) return ERROR;
                    if(TXMessage[j]==RXMessage[0]) Match=1; else return ERROR;
                    printf("Sent:'%c', Received:'%c', Match:%d\n",
TXMessage[j], RXMessage[0], Match);
                }
                break;
             case 1:
                printf("Ping Pong Baud Rate Test: UART#1 to UART#0\n");
                for(j=0; j<wcslen(TXMessage); j++)
                {
                    altera_16550_uart_write(uart_1, &TXMessage[j], 1, 0);
                    usleep(1000);
                    if(ERROR== altera_16550_uart_read(uart_0,  RXMessage, 1,
0)) return ERROR;
                    if(TXMessage[j]==RXMessage[0]) Match=1; else return ERROR;
                    printf("Sent:'%c', Received:'%c', Match:%d\n",
TXMessage[j], RXMessage[0], Match);
                }
                break;
             default:
                 break;
            }
            usleep(1000);
          }
  }
  free(UART0_Config);
  free(UART1_Config);
  return SUCCESS;
}

int UARTLineControlTest()
{
  UartConfig *UART0_Config = malloc(1*sizeof(UartConfig));
  UartConfig *UART1_Config = malloc(1*sizeof(UartConfig));

  int x=0, y=0, z=0, Match=0;
  const int nDataBit = 2, nParityBit=3, nStopBit=2;
```

```
  int DataBitCoverage[]= { /*CS_5, CS_6,*/ CS_7, CS_8};
  int ParityBitCoverage[]= {ODD_PARITY, EVEN_PARITY, NO_PARITY};
  int StopBitCoverage[]= {STOPB_1, STOPB_2};
  altera_16550_uart_state* uart_0;
  altera_16550_uart_state* uart_1;

  printf("============================== UART Line Control Test Starts Here
======================================\n");
  uart_0 = altera_16550_uart_open ("/dev/a_16550_uart_0");
  uart_1 = altera_16550_uart_open ("/dev/a_16550_uart_1");

  for(x=0; x<nStopBit; x++)
  {
      for (y=0; y<nParityBit; y++)
      {
          for (z=0; z<nDataBit; z++)
              {
                 UARTDefaultConfig(UART0_Config);
                UARTDefaultConfig(UART1_Config);
                UART0_Config->stop_bit=StopBitCoverage[x];
                UART1_Config->stop_bit=StopBitCoverage[x];
                UART0_Config->parity_bit=ParityBitCoverage[y];
                UART1_Config->parity_bit=ParityBitCoverage[y];
                UART0_Config->data_bit=DataBitCoverage[z];
                UART1_Config->data_bit=DataBitCoverage[z];

                printf("Testing : Stop Bit=%d, Data Bit=%d, Parity Bit=%d\n",
UART0_Config->stop_bit, UART0_Config->data_bit, UART0_Config->parity_bit);
                if(ERROR == alt_16550_uart_config (uart_0, UART0_Config))
return ERROR;
                if(ERROR == alt_16550_uart_config (uart_1, UART1_Config))
return ERROR;
                altera_16550_uart_write(uart_0, &TXMessage[0], 1, 0);
                usleep(1000);
                if(ERROR== altera_16550_uart_read(uart_1,  RXMessage, 1, 0))
return ERROR;
                if(TXMessage[0]==RXMessage[0]) Match=1; else
                    {
                    printf("Sent:'%c', Received:'%c', Match:%d\n",
TXMessage[0], RXMessage[0], Match);
                    return ERROR;
                    }
                printf("Sent:'%c', Received:'%c', Match:%d\n", TXMessage[0],
RXMessage[0], Match);
                }
      }
  }
  free(UART0_Config);
  free(UART1_Config);
  return SUCCESS;
}

int UARTFIFOModeTest()
{

  UartConfig *UART0_Config = malloc(1*sizeof(UartConfig));
  UartConfig *UART1_Config = malloc(1*sizeof(UartConfig));

  int i=0, direction=0, CharCounter=0, Match=0;
  const int nBaud = 2;
  int BaudRateCoverage[]= {BR115200, /*BR19200, BR38400, BR57600,*/ BR9600};
  altera_16550_uart_state* uart_0;
  altera_16550_uart_state* uart_1;

  printf("============================== UART FIFO Mode Test Starts Here
======================================\n");
  uart_0 = altera_16550_uart_open ("/dev/a_16550_uart_0");
  uart_1 = altera_16550_uart_open ("/dev/a_16550_uart_1");

  for (direction=0; direction<2; direction++)
  {
```

```
      for (i=0; i<nBaud; i++)
          {
              UARTDefaultConfig(UART0_Config);
            UARTDefaultConfig(UART1_Config);
            UART0_Config->baudrate=BaudRateCoverage[i];
            UART1_Config->baudrate=BaudRateCoverage[i];
            UART0_Config->fifo_mode = 1;
            UART1_Config->fifo_mode = 1;
              UART0_Config->hwfc = 0;
              UART1_Config->hwfc = 0;
            if(ERROR == alt_16550_uart_config (uart_0, UART0_Config)) return
ERROR;
            if(ERROR == alt_16550_uart_config (uart_1, UART1_Config)) return
ERROR;
            printf("Testing Baud Rate: %d\n", UART0_Config->baudrate);

            switch(direction)
            {
             case 0:
                printf("Ping Pong FIFO Test: UART#0 to UART#1\n");
                CharCounter=altera_16550_uart_write(uart_0, &TXMessage,
wcslen(TXMessage), 0);
                //usleep(50000);
                if(ERROR== altera_16550_uart_read(uart_1,  RXMessage,
wcslen(TXMessage), 0)) return ERROR;
                if(strcmp(TXMessage, RXMessage)==0) Match=1; else Match=0;
                printf("Sent:'%s' CharCount:%d, Received:'%s' CharCount:%d,
Match:%d\n", TXMessage, CharCounter, RXMessage, wcslen(RXMessage), Match);
                if(Match==0) return ERROR;
                break;
             case 1:
                printf("Ping Pong FIFO Test: UART#1 to UART#0\n");
                CharCounter=altera_16550_uart_write(uart_1, &TXMessage,
wcslen(TXMessage), 0);
                //usleep(50000);
                if(ERROR== altera_16550_uart_read(uart_0,  RXMessage,
wcslen(TXMessage), 0)) return ERROR;
                if(strcmp(TXMessage, RXMessage)==0) Match=1; else Match=0;
                printf("Sent:'%s' CharCount:%d, Received:'%s' CharCount:%d,
Match:%d\n", TXMessage, CharCounter, RXMessage, wcslen(RXMessage), Match);
                if(Match==0) return ERROR;
                break;
             default:
                 break;
            }
            //usleep(100000);
          }
  }
  free(UART0_Config);
  free(UART1_Config);
  return SUCCESS;
}

int main()
{
  int result=0;

  result = UARTBaudRateTest();
  if(result==ERROR)
  {
   printf("UARTBaudRateTest FAILED\n");
   return ERROR;
  }

  result = UARTLineControlTest();
  if(result==ERROR)
  {
   printf("UARTLineControlTest FAILED\n");
   return ERROR;
  }
```

```
  result = UARTFIFOModeTest();
  if(result==ERROR)
  {
    printf("UARTFIFOModeTest FAILED\n");
    return ERROR;
  }
  printf("\n\nALL TESTS PASS\n\n");
  return 0;
}
```

10.4. Address Map and Register Descriptions

Table 74. altr_uart_csr Address Map

Register	Offset	Width	Access	Reset Value	Description
rbr_thr_dll	0x0	32	RW	0x00000000	Rx Buffer, Tx Holding, and Divisor Latch Low
ier_dlh	0x4	32	RW	0x00000000	Interrupt Enable and Divisor Latch High
iir	0x8	32	R	0x00000001	Interrupt Identity Register (when read)
fcr	0x8	32	W	0x00000000	FIFO Control (when written)
lcr	0xC	32	RW	0x00000000	Line Control Register
mcr	0x10	32	RW	0x00000000	Modem Control Register
lsr	0x14	32	R	0x00000060	Line Status Register
msr	0x18	32	R	0x00000000	Modem Status Register
scr	0x1C	32	RW	0x00000000	Scratchpad Register
afr	0x100	32	RW	0x00000000	Additional Features Register
tx_low	0x104	32	RW	0x00000000	Transmit FIFO Low Watermark Register

Note: RC-Read to Clear

10.4.1. rbr_thr_dll

Identifier	Title	Offset	Access	Reset Value	Description
rbr_thr_dll	Rx Buffer, Tx Holding, and Divisor Latch Low	0x0	RW	0x00000000	This is a multi-function register. This register holds receives and transmit data and controls the least-signficant 8 bits of the baud rate divisor.

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
-															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-								rbr_thr_dll							

Table 75. rbr_thr_dll Fields

Bit	Name/Identifier	Description	Access	Reset
[31:8]	-	Reserved	R	0x0
[7:0]	rbr_thr_dll	• Receive Buffer Register: This register contains the data byte received on the serial input port (sin). The data in this register is valid only if the Data Ready (LSR[0] is set to 1). If FIFOs are disabled (FCR[0] is cleared to 0) the data in the RBR must be read before the next data arrives, otherwise it will be overwritten, resulting in an overrun error. If FIFOs are enabled (FCR[0] set to 1) this register accesses the head of the receive FIFO. If the receive FIFO is full, and this register is not read before the next data character arrives, then the data already in the FIFO will be preserved but any incoming data will be lost. An overrun error will also occur. • Transmit Holding Register: This register contains data to be transmitted on the serial output port (sout). Data should only be written to the THR when the THR Empty bit (LSR[5] is set to 1). If FIFOs are disabled (FCR[0] is set to 0) and THRE is set to 1, writing a single character to the THR clears the THRE. Any additional writes to the THR before the THRE is set again causes the THR data to be overwritten. If FIFO's are enabled (FCR[0] is set to 1) and THRE is set, the FIFO can be filled up to a pre-configured depth (FIFO_DEPTH). Any attempt to write data when the FIFO is full results in the write data being lost. • Divisor Latch Low: This register makes up the lower 8-bits of a 16-bit, Read/write, Divisor Latch register that contains the baud rate divisor for the UART. This register may only be accessed when the DLAB bit (LCR[7] is set to 1). The output baud rate is equal to the system clock (clk) frequency divided by sixteen times the value of the baud rate divisor, as follows: baud rate = (system clock freq) / (16 * divisor) *Note:* With the Divisor Latch Registers (DLL and DLH) set to zero, the baud clock is disabled and no serial communications will occur. Also, once the DLL is set, at least 8 system clock cycles should be allowed to pass before transmitting or receiving data.	RW	0x00

10.4.2. ier_dlh

Identifier	Title	Offset	Access	Reset Value	Description
ier_dlh	Interrupt Enable and Divisor Latch High	0x4	RW	0x00000000	The ier_dlh (Interrupt Enable Register) may only be accessed when the DLAB bit [7] of the LCR Register is set to 0. Allows control of the Interrupt Enables for transmit and receive functions.This is a multi-function register. This register enables/disables receive and transmit interrupts and also controls the most-significant 8-bits of the baud rate divisor. The Divisor Latch High Register is accessed when the DLAB bit (LCR[7] is set to 1). Bits[7:0] contain the high order 8-bits of the baud rate divisor. The output baud rate is equal to the system clock (clk) frequency divided by sixteen times the value of the baud rate divisor, as follows: baud rate = (system clock freq) / (16 * divisor) *Note:* With the Divisor Latch Registers (DLL and DLH) set to zero, the baud clock is disabled and no serial communications will occur. Also, once the DLL is set, at least 8 system clock cycles should be allowed to pass before transmitting or receiving data.

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
-															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-								dlh7_4				edssi_dhl3	elsi_dhl2	etbei_dlh1	erbfi_dlh0

Table 76. ier_dlh Fields

Bit	Name/Identifier	Description	Access	Reset
[31:8]	-	Reserved	R	0x0
[7:4]	DLH[7:4] (dlh7_4)	• Divisor Latch High Register: Bit 4, 5, 6 and 7 of DLH value.	RW	0x0
[3]	DLH[3] and Enable Modem Status Interrupt (edssi_dhl3)	• Divisor Latch High Register: Bit 3 of DLH value. • Interrupt Enable Register: This is used to enable/disable the generation of Modem Status Interrupts. This is the fourth highest priority interrupt.	RW	0x0

continued...

Bit	Name/Identifier	Description	Access	Reset
[2]	DLH[2] and Enable Receiver Line Status (`elsi_dhl2`)	• Divisor Latch High Register: Bit 2 of DLH value. • Interrupt Enable Register: This is used to enable/disable the generation of Receiver Line Status Interrupt. This is the highest priority interrupt	RW	0x0
[1]	DLH[1] and Transmit Data Interrupt Control (`etbei_dlh1`)	• Divisor Latch High Register: Bit 1 of DLH value. • Interrupt Enable Register: Enable Transmit Holding Register Empty Interrupt. This is used to enable/disable the generation of Transmitter Holding Register Empty Interrupt. This is the third highest priority interrupt.	RW	0x0
[0]	DLH[0] and Receive Data Interrupt Enable (`erbfi_dlh0`)	• Divisor Latch High Register: Bit 0 of DLH value. • Interrupt Enable Register: This is used to enable/disable the generation of the Receive Data Available Interrupt and the Character Timeout Interrupt (if FIFO's enabled). These are the second highest priority interrupts.	RW	0x0

10.4.3. iir

Identifier	Title	Offset	Access	Reset Value	Description
iir	Interrupt Identity Register	0x8	R	0x00000001	Returns interrupt identification and FIFO enable/disable when read.

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
-															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-								fifose		-		id			

Table 77. iir Fields

Bit	Name/Identifier	Description	Access	Reset
[31:8]	-	Reserved	R	0x0
[7:6]	FIFOs Enabled (fifose)	The FIFOs Enabled is used to indicate whether the FIFO's are enabled or disabled.	R	0x0
[5:4]	-	Reserved	R	0x0
[3:0]	Interrupt ID (id)	The Interrupt ID indicates the highest priority pending interrupt. Refer to the Table 78 on page 118 table below for more details.	R	0x1

Table 78. Interrupt Priority

IIR ID	Interrupt	Priority
4'b0000	Modem status	5th
4'b0001	No interrupt pending	6th
4'b0010	THR empty (reflect TX_Low empty threshold if ARF[0] is '1)	4th
4'b0100	Received data available	2nd
4'b0110	Receiver line status	1st
4'b1100	Character timeout	3rd

10.4.4. fcr

Identifier	Title	Offset	Access	Reset Value	Description
fcr	FIFO Control	0x8	W	0x00000000	Controls FIFO operation when written.

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
-															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-								rt		-		dmam	xfifor	rfifor	fifoe

Table 79. fcr Fields

Bit	Name/Identifier	Description	Access	Reset
[31:8]	-	Reserved	R	0x0
[7:6]	Rx Trigger Level (rt)	This register is configured to implement FIFOs RxTrigger (or RT). This is used to select the trigger level in the receiver FIFO at which the Received Data Available Interrupt will be generated. In auto flow control mode it is used to determine when the rts_n signal will be de-asserted The following trigger levels are supported: • 00 - One character in FIFO • 01 - FIFO 1/4 full • 10 - FIFO 1/2 ful • 11 - FIFO two less than full	W	0x0
[5:4]	-	Reserved	R	0x0
[3]	DMA Mode (dmam)	This determines the DMA signaling mode used for the uart_dma_tx_req_n and uart_dma_rx_req_n output signals when additional DMA handshaking signals are not selected. DMA mode 0 supports single DMA data transfers at a time. In mode 0, the uart_dma_tx_req_n signal goes active low under the following conditions: • When the Transmitter Holding Register is empty in non-FIFO mode. • When the transmitter FIFO is empty in FIFO mode. It goes inactive under the following conditions: • When a single character has been written into the Transmitter Holding Register or transmitter FIFO. • When the transmitter FIFO is above the threshold. DMA mode 1 supports multi-DMA data transfers, where multiple transfers are made continuously until the receiver FIFO has been emptied or the transmit FIFO has been filled. In mode 1 the uart_dma_tx_req_n signal is asserted under the following condition: • When the transmitter FIFO is empty.	W	0x0

continued...

Bit	Name/Identifier	Description	Access	Reset
[2]	Tx FIFO Reset (xfifor)	This bit resets the control portion of the transmit FIFO and treats the FIFO as empty. Note that this bit is 'self-clearing' and it is not necessary to clear this bit. Please allow for 8 clock cycles to pass after changing this register bit before reading from RBR or writing to THR.	W	0x0
[1]	Rx FIFO Reset (rfifor)	Resets the control portion of the receive FIFO and treats the FIFO as empty. Note that this bit is self-clearing' and it is not necessary to clear this bit. Allow for 8 clock cycles to pass after changing this register bit before reading from RBR or writing to THR.	W	0x0
[0]	FIFO Enable (fifoe)	This bit enables/disables the transmit (Tx) and receive (Rx) FIFO's. Whenever the value of this bit is changed both the Tx and Rx controller portion of FIFO's will be reset. Any existing data in both Tx and Rx FIFO will be lost when this bit is changed. Please allow for 8 clock cycles to pass after changing this register bit before reading from RBR or writing to THR.	W	0x0

10.4.5. lcr

Identifier	Title	Offset	Access	Reset Value	Description
lcr	Line Control Register	0xC	RW	0x00000000	Formats serial data.

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
-															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-							dls9	dlab	break	sp	eps	pen	stop	dls	

Table 80. lcr Fields Description

Bit	Name/Identifier	Description	Access	Reset
[31:9]	-	Reserved	R	0x0
[8]	Data Length Select (dls9)	Issue 1'b1 to LCR[8] and 2'b00 to LCR[1:0] to turn on 9 data bits per character that the peripheral will transmit and receive.	RW	0x0
[7]	Divisor Latch Access Bit (dlab)	This is used to enable reading and writing of the Divisor Latch register (DLL and DLH) to set the baud rate of the UART. This bit must be cleared after initial baud rate setup in order to access other registers.	RW	0x0
[6]	Break Control Bit (break)	This is used to cause a break condition to be transmitted to the receiving device. If set to one the serial output is forced to the spacing (logic 0) state until the Break bit is cleared.	RW	0x0
[5]	Stick Parity (sp)	The SP bit works in conjunction with the EPS and PEN bits. When odd parity is selected (EPS = 0), the PARITY bit is transmitted and checked as set. When even parity is selected (EPS = 1), the PARITY bit is transmitted and checked as cleared.	RW	0x0
[4]	Even Parity Select (eps)	This is used to select between even and odd parity, when parity is enabled (PEN set to one). If set to one, an even number of logic '1's is transmitted or checked. If set to zero, an odd number of logic '1's is transmitted or checked.	RW	0x0
[3]	Parity Enable (pen)	This bit is used to enable and disable parity generation and detection in a transmitted and received data character.	RW	0x0
[2]	Stop Bits (stop)	Number of stop bits. This is used to select the number of stop bits per character that the peripheral will transmit and receive. Note that regardless of the number of stop bits selected the receiver will only check the first stop bit.	RW	0x0

Data Bits	LCR[2] = 0	LCR[2] = 1
5	1	1.5
6	1	2
7	1	2
8	1	2

continued...

Bit	Name/Identifier	Description	Access	Reset
		• If bit 2 is a logic 0, one stop bit is generated in the transmitted data. • If bit 2 is a logic 1 when a 5-bit word length is selected through bits 0 and 1, one and a half stop bits are generated. • If bit 2 is a logic 1 when either a 6-, 7-, or 8-bit word length is selected, two stop bits are generated. The Receiver checks the first stop-bit only, regardless of the number of stop bits selected.		
[1:0]	Data Length Select (dls)	Selects the number of data bits per character that the peripheral will transmit and receive. • 0 - 5 data bits per character • 1 - 6 data bits per character • 2 - 7 data bits per character • 3 - 8 data bits per character	RW	0x0

10.4.6. mcr

Identifier	Title	Offset	Access	Reset Value	Description
mcr	Modem Control Register	0x10	RW	0x00000000	Reports various operations of the modem signals.

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
-															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-										afce	loopback	out2	out1	rts	dtr

Table 81. mcr Fields Descriptions

Bit	Name/Identifier	Description	Access	Reset
[31:6]	-	Reserved	R	0x0
[5]	Hardware Auto Flow Control Enable (afce)	When FIFOs are enabled (FCR[0]), the Auto Flow Control enable bits are active. This enabled UART to dynamically assert and deassert rts_n based on Receive FIFO trigger level	RW	0x0
[4]	LoopBack Bit (loopback)	This is used to put the UART into a diagnostic mode for test purposes. If UART mode is NOT active, bit [6] of the modem control register MCR is set to zero, data on the sout line is held high, while serial data output is looped back to the sin line, internally. In this mode all the interrupts are fully functional. Also, in loopback mode, the modem control inputs (dsr_n, cts_n, ri_n, dcd_n) are disconnected and the modem control outputs (dtr_n, rts_n, out1_n, out2_n) are looped-back to the inputs, internally.	RW	0x0
[3]	Out2 (out2)	This is used to directly control the user-designated out2_n output. The value written to this location is inverted and driven out on out2_n	RW	0x0
[2]	Out1 (out1)	This is used to directly control the user-designated out1_n output. The value written to this location is inverted and driven out on out1_n pin.	RW	0x0
[1]	Request to Send (rts)	This is used to directly control the Request to Send (rts_n) output. The Request to Send (rts_n) output is used to inform the modem or data set that the UART is ready to exchange data. When Auto RTS Flow Control is not enabled (MCR[5] set to zero), the rts_n signal is set low by programming this register to a high. If Auto Flow Control is active (MCR[5] set to 1) and FIFO's enable (FCR[0] set to 1), the rts_n output is controlled in the same way, but is also gated with the receiver FIFO threshold trigger (rts_n is inactive high when above the threshold). The rts_n signal will be de-asserted when this register is set low.	RW	0x0
[0]	Data Terminal Ready (dtr)	This is used to directly control the Data Terminal Ready output. The value written to this location is inverted and driven out on uart_dtr_n. The Data Terminal Ready output is used to inform the modem or data set that the UART is ready to establish communications.	RW	0x0

10.4.7. lsr

Identifier	Title	Offset	Access	Reset Value	Description
lsr	Line Status Register	0x14	R	0x00000060	Reports status of transmit and receive.

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
-															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-								rfe	temt	thre	bi	fe	pe	oe	dr

Table 82. lsr Fields

Bit	Name/ Identifier	Description	Access	Reset
[31:8]	-	Reserved	R	0x0
[7]	Receiver FIFO Error bit (rfe)	This bit is only relevant when FIFO's are enabled (FCR[0] set to one). This is used to indicate if there is at least one parity error, framing error, or break indication in the FIFO. This bit is cleared when the LSR is read and the character with the error is at the top of the receiver FIFO and there are no subsequent errors in the FIFO.	R	0x0
[6]	Transmitter Empty bit (temt)	If in FIFO mode and FIFO's enabled (FCR[0] set to one), this bit is set whenever the Transmitter Shift Register and the FIFO are both empty. If FIFO's are disabled, this bit is set whenever the Transmitter Holding Register and the Transmitter Shift Register are both empty. Indicator is cleared when new data is written into the THR or Transmit FIFO.	R	0x1
[5]	Transmit Holding Register Empty bit (thre)	This bit indicates that the THR or Tx FIFO is empty. This bit is set when data is transferred from the THR or Tx FIFO to the transmitter shift register and no new data has been written to the THR or Tx FIFO. This also causes a THRE Interrupt to execute, if the THRE Interrupt is enabled.	R	0x1
[4]	Break Interrupt (bi)	This is used to indicate the detection of a break sequence on the serial input data. Set whenever the serial input, sin, is held in a logic 0 state for longer than the sum of start time + data bits + parity + stop bits. A break condition on serial input causes one and only one character, consisting of all zeros, to be received by the UART. The character associated with the break condition is carried through the FIFO and is revealed when the character is at the top of the FIFO. This bit always stays in sync with the associated character in RBR. If the current associated character is read through RBR, this bit will be updated to be in sync with the next character in RBR. Reading the LSR clears the BI bit.	RC	0x0
[3]	Framing Error (fe)	This is used to indicate the occurrence of a framing error in the receiver. A framing error occurs when the receiver does not detect a valid STOP bit in the received data. In the FIFO mode, since the framing error is associated with a character received, it is revealed when the character with the framing error is at the top of the FIFO. When a framing error occurs the UART will try to resynchronize. It does this by assuming that the error was due to the start bit of the next character and then continues receiving the	RC	0x0

continued...

Bit	Name/ Identifier	Description	Access	Reset
		other bit data, and/or parity and stop. It should be noted that the Framing Error (FE) bit(LSR[3]) will be set if a break interrupt has occurred, as indicated by a Break Interrupt BIT bit (LSR[4]). This bit always stays in sync with the associated character in RBR. If the current associated character is read through RBR, this bit will be updated to be in sync with the next character in RBR. Reading the LSR clears the FE bit.		
[2]	Parity Error (pe)	This is used to indicate the occurrence of a parity error in the receiver if the Parity Enable (PEN) bit (LCR[3]) is set. Since the parity error is associated with a character received, it is revealed when the character with the parity error arrives at the top of the FIFO. It should be noted that the Parity Error (PE) bit (LSR[2]) will be set if a break interrupt has occurred, as indicated by Break Interrupt (BI) bit (LSR[4]). In this situation, the Parity Error bit is set depending on the combination of EPS (LCR[4]) and DLS (LCR[1:0]). This bit always stays in sync with the associated character in RBR. If the current associated character is read through RBR, this bit will be updated to be in sync with the next character in RBR. Reading the LSR clears the PE bit.	RC	0x0
[1]	Overrun error bit (oe)	This is used to indicate the occurrence of an overrun error. This occurs if a new data character was received before the previous data was read. In the non-FIFO mode, the OE bit is set when a new character arrives in the receiver before the previous character was read from the RBR. When this happens, the data in the RBR is overwritten. In the FIFO mode, an overrun error occurs when the FIFO is full and new character arrives at the receiver. The data in the FIFO is retained and the data in the receive shift register is lost.Reading the LSR clears the OE bit.	RC	0x0
[0]	Data Ready bit (dr)	This is used to indicate that the receiver contains at least one character in the RBR or the receiver FIFO. This bit is cleared when the RBR is read in the non-FIFO mode, or when the receiver FIFO is empty, in the FIFO mode.	R	0x0

10.4.8. msr

Identifier	Title	Offset	Access	Reset Value	Description
msr	Modem Status Register	0x18	R	0x00000000	It should be noted that whenever bits 0, 1, 2 or 3 are set to logic one, to indicate a change on the modem control inputs, a modem status interrupt will be generated if enabled via the IER regardless of when the change occurred. Since the delta bits (bits 0, 1, 3) can get set after a reset if their respective modem signals are active (see individual bits for details), a read of the MSR after reset can be performed to prevent unwanted interrupts.

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
-															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-								dcd	ri	dsr	cts	ddcd	teri	ddsr	dcts

Table 83. msr Fields

Bit	Name/Identifier	Description	Access	Reset
[31:8]	-	Reserved	R	0x0
[7]	Data Carrier Detect (dcd)	This bit is the complement of the modem control line (dcd_n). This bit is used to indicate the current state of dcd_n. When the Data Carrier Detect input (dcd_n) is asserted it is an indication that the carrier has been detected by the modem or data set.	R	0x0
[6]	Ring Indicator (ri)	This bit is the complement of modem control line (ri_n). This bit is used to indicate the current state of ri_n. When the Ring Indicator input (ri_n) is asserted it is an indication that a telephone ringing signal has been received by the modem or data set.	R	0x0
[5]	Data Set Ready (dsr)	This bit is the complement of modem control line dsr_n. This bit is used to indicate the current state of dsr_n. When the Data Set Ready input (dsr_n) is asserted it is an indication that the modem or data set is ready to establish communications with the uart.	R	0x0
[4]	Clear to Send (cts)	This bit is the complement of modem control line cts_n. This bit is used to indicate the current state of cts_n. When the Clear to Send input (cts_n) is asserted it is an indication that the modem or data set is ready to exchange data with the uart.	R	0x0
[3]	Delta Data Carrier Detect (ddcd)	This is used to indicate that the modem control line dcd_n has changed since the last time the MSR was read. Reading the MSR clears the DDCD bit.	RC	0x0

continued...

Bit	Name/Identifier	Description	Access	Reset
		Note: If the DDCD bit is not set and the dcd_n signal is asserted (low) and a reset occurs (software or otherwise), then the DDCD bit will get set when the reset is removed if the dcd_n signal remains asserted.		
[2]	Trailing Edge of Ring Indicator (teri)	This is used to indicate that a change on the input ri_n (from an active low, to an inactive high state) has occurred since the last time the MSR was read. Reading the MSR clears the TERI bit.	RC	0x0
[1]	Delta Data Set Ready (ddsr)	This is used to indicate that the modem control line dsr_n has changed since the last time the MSR was read. Reading the MSR clears the DDSR bit. *Note:* If the DDSR bit is not set and the dsr_n signal is asserted (low) and a reset occurs (software or otherwise), then the DDSR bit will get set when the reset is removed if the dsr_n signal remains asserted.	RC	0x0
[0]	Delta Clear to Send (dcts)	This is used to indicate that the modem control line cts_n has changed since the last time the MSR was read. Reading the MSR clears the DCTS bit. *Note:* If the DCTS bit is not set and the cts_n signal is asserted (low) and a reset occurs (software or otherwise), then the DCTS bit will get set when the reset is removed if the cts_n signal remains asserted.	RC	0x0

10.4.9. scr

Identifier	Title	Offset	Access	Reset Value	Description
scr	Scratchpad Register	0x1C	RW	0x00000000	Scratchpad Register

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
-															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-								scr							

Table 84. scr Fields

Bit	Name	Description	Access	Reset
[31:8]	-	Reserved	R	0x0
[7:0]	Scratchpad Register (scr)	This register is for programmers to use as a temporary storage space.	RW	0x0

10.4.10. afr

Identifier	Title	Offset	Access	Reset Value	Description
afr	Additional Features Register	0x100	RW	0x00000000	These registers enable additional features in the soft UART controller. These features are specific to Intel FPGA.

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
-															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-															tx_low_en

Table 85. afr Fields

Bit	Name/Identifier	Description	Access	Reset
[31:1]	-	Reserved	R	0x0
[0]	Transmit FIFO Low Watermark Enable Register (tx_low_en)	This bit controls the Tx FIFO Low Watermark feature. This feature requires FIFO to be enabled (FCR[0]). When enabled, the UART will send a Transmit Holding Register Empty status interrupt when the Transmit FIFO level is at or below the value stored in tx_low. Legal values for tx_low can range from zero up to depth of FIFO minus two. UART behavior is undefined when tx_low is set to illegal values. • 1 - Transmit FIFO Low Watermark is set by tx_low • 0 - Transmit FIFO Low Watermark is unset This value must only be changed when the Transmit FIFO is empty or before FIFO is enabled (FCR[0]). This register is meant to be changed during UART initialization before active traffic is sent. The Transmit FIFO should be reset using FCR[2] after any changes to this value.	RW	0x0

10.4.11. tx_low

Identifier	Title	Offset	Access	Reset Value	Description
tx_low	Transmit FIFO Low Watermark Register	ox104	RW	0x00000000	This register is used to set the value of the Transmit FIFO Low Watermark.

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
-															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-							value								

Table 86. tx_low Fields

Bit	Name/Identifier	Description	Access	Reset
[31:9]	-	Reserved	R	0x0
[8:0]	Transmit FIFO Low Watermark (value)	Set the Transmit FIFO Low Watermark Value. The lowest legal value is zero The highest legal value is two less than the FIFO Depth This value must only be changed when the Transmit FIFO is empty or before FIFO is enabled (FCR[0]).	RW	0x00

10.5. Intel FPGA 16550 Compatible UART Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2021.03.29	21.1	Added support for Intel eASIC N5X devices.
2019.07.16	19.1	Updated the description of stop bits in *Line Control Register*.
2018.05.07	18.0	• Updated description of FIFO_DEPTH parameter. • Updated *Figure: Hardware Auto Flow-Control Between two UARTs*.

Date	Version	Changes
November 2017	2017.11.06	Removed the minimum clock requirement in the *Table: Clock and Reset Signal Interface*.
October 2016	2016.10.28	Two new registers: • afr on page 129 • tx_low on page 130 Updated: • lsr on page 124 Bit [5] • fcr on page 119 Bit [7:6] • New table added to iir on page 118 section
December 2015	2015.12.16	Product ID changed in "16550 UART Release Information" section.
November 2015	2015.11.06	Updated the following topics:

continued...

Date	Version	Changes
		• Core Overview on page 95 • Feature Description — Table 54 on page 95 • General Architecture — Figure 31 on page 98 • Configuration Parameters — Table 61 on page 100 • DMA Support on page 100 • Supported Features — Table 65 on page 105 • Configuration — Figure 38 on page 106 • UART Device Structure on page 108 — Example 1 and 2 • Address Map and Register Descriptions on page 114
June 2015	2015.06.12	• Added "16550 UART General Programming Flow Chart" section • Added "16550 UART Release Information" section • Added "Address Map and Register Descriptions" section • Added Stick parity/Force parity feature into the "UART Features and Configurability" table in the "Feature Description" section • Updated "Interface" section with `sout_oe` signal details in the "Flow Control" table • Updated "Underrun" section
July 2014	2014.07.24	Initial Release.

11. UART Core

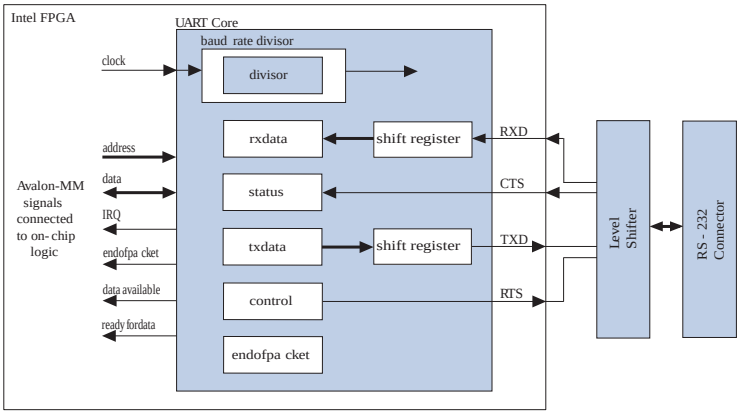
11.1. Core Overview

The UART core with Avalon interface implements a method to communicate serial character streams between an embedded system on an Intel FPGA and an external device. The core implements the RS-232 protocol timing, and provides adjustable baud rate, parity, stop, and data bits. The feature set is configurable, allowing designers to implement just the necessary functionality for a given system.

The core provides an Avalon Memory-Mapped (Avalon-MM) agent interface that allows Avalon-MM host peripherals (such as a Nios II processor) to communicate with the core simply by reading and writing control and data registers.

11.2. Functional Description

Figure 39. Block Diagram of the UART Core in a Typical System



The core has two user-visible parts:

- The register file, which is accessed via the Avalon-MM agent port
- The RS-232 signals, RXD, TXD, CTS, and RTS

11.2.1. Avalon-MM Agent Interface and Registers

The UART core provides an Avalon-MM agent interface to the internal register file. The user interface to the UART core consists of six, 16-bit registers: `control`, `status`, `rxdata`, `txdata`, `divisor`, and `endofpacket`. A host peripheral, such as a Nios II processor, accesses the registers to control the core and transfer data over the serial connection.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

The UART core provides an active-high interrupt request (IRQ) output that can request an interrupt when new data has been received, or when the core is ready to transmit another character. For further details, refer to the **Interrupt Behavior** section.

For more information, refer to **Interval Timer Core** section.

For details about the Avalon-MM interface, refer to the Avalon Interface Specifications.

11.2.2. RS-232 Interface

The UART core implements RS-232 asynchronous transmit and receive logic. The UART core sends and receives serial data via the TXD and RXD ports. The I/O buffers on most Intel FPGA families do not comply with RS-232 voltage levels, and may be damaged if driven directly by signals from an RS-232 connector. To comply with RS-232 voltage signaling specifications, an external level-shifting buffer is required (for example, Maxim MAX3237) between the FPGA I/O pins and the external RS-232 connector.

The UART core uses a logic 0 for mark, and a logic 1 for space. An inverter inside the FPGA can be used to reverse the polarity of any of the RS-232 signals, if necessary.

11.2.3. Transmitter Logic

The UART transmitter consists of a 7-, 8-, or 9-bit `txdata` holding register and a corresponding 7-, 8-, or 9-bit transmit shift register. Avalon-MM host peripherals write the `txdata` holding register via the Avalon-MM agent port. The transmit shift register is loaded from the `txdata` register automatically when a serial transmit shift operation is not currently in progress. The transmit shift register directly feeds the TXD output. Data is shifted out to TXD LSB first.

These two registers provide double buffering. A host peripheral can write a new value into the `txdata` register while the previously written character is being shifted out. The host peripheral can monitor the transmitter's status by reading the `status` register's transmitter ready (TRDY), transmitter shift register empty (`tmt`), and transmitter overrun error (TOE) bits.

The transmitter logic automatically inserts the correct number of start, stop, and parity bits in the serial TXD data stream as required by the RS-232 specification.

11.2.4. Receiver Logic

The UART receiver consists of a 7-, 8-, or 9-bit receiver-shift register and a corresponding 7-, 8-, or 9-bit `rxdata` holding register. Avalon-MM host peripherals read the `rxdata` holding register via the Avalon-MM agent port. The `rxdata` holding register is loaded from the receiver shift register automatically every time a new character is fully received.

These two registers provide double buffering. The `rxdata` register can hold a previously received character while the subsequent character is being shifted into the receiver shift register.

A host peripheral can monitor the receiver's status by reading the `status` register's read-ready (RRDY), receiver-overrun error (ROE), break detect (BRK), parity error (PE), and framing error (FE) bits. The receiver logic automatically detects the correct

number of start, stop, and parity bits in the serial RXD stream as required by the RS-232 specification. The receiver logic checks for four exceptional conditions, frame error, parity error, receive overrun error, and break, in the received data and sets corresponding `status` register bits.

11.2.5. Baud Rate Generation

The UART core's internal baud clock is derived from the Avalon-MM clock input. The internal baud clock is generated by a clock divider. The divisor value can come from one of the following sources:

- A constant value specified at system generation time
- The 16-bit value stored in the `divisor` register

The `divisor` register is an optional hardware feature. If it is disabled at system generation time, the divisor value is fixed and the baud rate cannot be altered.

11.3. Instantiating the Core

Instantiating the UART in hardware creates at least two I/O ports for each UART core: An RXD input, and a TXD output. The following sections describe the available options.

11.3.1. Configuration Settings

This section describes the configuration settings.

11.3.1.1. Baud Rate Options

The UART core can implement any of the standard baud rates for RS-232 connections. The baud rate can be configured in one of two ways:

- **Fixed rate**—The baud rate is fixed at system generation time and cannot be changed via the Avalon-MM agent port.
- **Variable rate**—The baud rate can vary, based on a clock divisor value held in the `divisor` register. A host peripheral changes the baud rate by writing new values to the `divisor` register.

  The baud rate is calculated based on the clock frequency provided by the Avalon-MM interface. Changing the system clock frequency in hardware without regenerating the UART core hardware results in incorrect signaling.

The baud rate is calculated based on the clock frequency provided by the Avalon-MM interface. Changing the system clock frequency in hardware without regenerating the UART core hardware results in incorrect signaling.

11.3.1.2. Baud Rate (bps) Setting

The **Baud Rate** setting determines the default baud rate after reset. The **Baud Rate** option offers standard preset values.

The baud rate value is used to calculate an appropriate clock divisor value to implement the desired baud rate. Baud rate and divisor values are related as shown in the follow two equations:

Divisor Formula:

$$Divisor = \left(\frac{clock\ frequency}{baud\ rate}\right) - 1$$

Baud rate Formula:

$$Baud\ rate = \frac{clock\ frequency}{divisor + 1}$$

11.3.1.3. Baud Rate Can Be Changed By Software Setting

When this setting is on, the hardware includes a 16-bit `divisor` register at address offset 4. The `divisor` register is writable, so the baud rate can be changed by writing a new value to this register.

When this setting is off, the UART hardware does not include a `divisor` register. The UART hardware implements a constant baud divisor, and the value cannot be changed after system generation. In this case, writing to address offset 4 has no effect, and reading from address offset 4 produces an undefined result.

11.3.1.4. Data Bits, Stop Bits, Parity

The UART core's parity, data bits and stop bits are configurable. These settings are fixed at system generation time; they cannot be altered via the register file.

Table 87. Data Bits Settings

Setting	Legal Values	Description
Data Bits	7, 8, 9	This setting determines the widths of the `txdata`, `rxdata`, and `endofpacket` registers.
Stop Bits	1, 2	This setting determines whether the core transmits 1 or 2 stop bits with every character. The core always terminates a receive transaction at the first stop bit, and ignores all subsequent stop bits, regardless of this setting.
Parity	None, Even, Odd	This setting determines whether the UART core transmits characters with parity checking, and whether it expects received characters to have parity checking. When **Parity** is set to **None**, the transmit logic sends data without including a parity bit, and the receive logic presumes the incoming data does not include a parity bit. The PE bit in the `status` register is not implemented; it always reads 0. When **Parity** is set to **Odd** or **Even**, the transmit logic computes and inserts the required parity bit into the outgoing TXD bitstream, and the

continued...

Setting	Legal Values	Description
		receive logic checks the parity bit in the incoming RXD bitstream. If the receiver finds data with incorrect parity, the PE bit in the `status` register is set to 1. When **Parity** is **Even**, the parity bit is 0 if the character has an even number of 1 bits; otherwise the parity bit is 1. Similarly, when parity is **Odd**, the parity bit is 0 if the character has an odd number of 1 bits.

11.3.1.5. Synchronizer Stages

The option **Synchronizer Stages** allows you to specify the length of synchronization register chains. These register chains are used when a metastable event is likely to occur and the length specified determines the meantime before failure. The register chain length, however, affects the latency of the core.

For more information on metastability in Intel FPGA devices, refer to Understanding Metastability in FPGAs.

For more information on metastability analysis and synchronization register chains, refer to the Area and Timing Optimization chapter in volume 2 of the *Intel Quartus Prime Handbook*.

11.3.1.6. Streaming Data (DMA) Control

The UART core can also optionally include the end-of-packet register.

11.3.1.6.1. Include End-of-Packet Register

When this setting is on, the UART core includes:

- A 7-, 8-, or 9-bit `endofpacket` register at address-offset 5. The data width is determined by the **Data Bits** setting.
- EOP bit in the `status` register.
- IEOP bit in the `control` register.
- `endofpacket` signal.

EOP detection can be used with a DMA controller, for example, to implement a UART that automatically writes received characters to memory until a specified character is encountered in the incoming RXD stream. The terminating (EOP) character's value is determined by the `endofpacket` register.

When the EOP register is disabled, the UART core does not include the EOP resources. Writing to the `endofpacket` register has no effect, and reading produces an undefined value.

11.4. Software Programming Model

The following sections describe the software programming model for the UART core, including the register map and software declarations to access the hardware. For Nios II processor users, Intel provides hardware abstraction layer (HAL) system library drivers that enable you to access the UART core using the ANSI C standard library functions, such as `printf()` and `getchar()`.

11.4.1. HAL System Library Support

The Intel-provided driver implements a HAL character-mode device driver that integrates into the HAL system library for Nios II systems. HAL users should access the UART via the familiar HAL API and the ANSI C standard library, rather than accessing the UART registers. `ioctl()` requests are defined that allow HAL users to control the hardware-dependent aspects of the UART.

Note: If your program uses the HAL device driver to access the UART hardware, accessing the device registers directly interferes with the correct behavior of the driver.

For Nios II processor users, the HAL system library API provides complete access to the UART core's features. Nios II programs treat the UART core as a character mode device, and send and receive data using the ANSI C standard library functions.

The driver supports the CTS/RTS control signals when they are enabled in Platform Designer. Refer to **Driver Options: Fast Versus Small Implementations** section.

The following code demonstrates the simplest possible usage, printing a message to `stdout` using `printf()`. In this example, the system contains a UART core, and the HAL system library has been configured to use this device for `stdout`.

Example 4. Example: Printing Characters to a UART Core as stdout

```
#include <stdio.h>
int main ()
{
printf("Hello world.\n");
return 0;
}
```

The following code demonstrates reading characters from and sending messages to a UART device using the C standard library. In this example, the system contains a UART core named `uart1` that is not necessarily configured as the `stdout` device. In this case, the program treats the device like any other node in the HAL file system.

For more information about the HAL system library, refer to the *Nios II Software Developer's Handbook*.

Example 5. Example: Sending and Receiving Characters

```
/* A simple program that recognizes the characters 't' and 'v' */
#include <stdio.h>
#include <string.h>
int main ()
{
 char* msg = "Detected the character 't'.\n";
 FILE* fp;
 char prompt = 0;
 fp = fopen ("/dev/uart1", "r+"); //Open file for reading and writing
 if (fp)
```

```
 {
 while (prompt != 'v')
 { // Loop until we receive a 'v'.
 prompt = getc(fp); // Get a character from the UART.
 if (prompt == 't')
 { // Print a message if character is 't'.
 fwrite (msg, strlen (msg), 1, fp);
 }
 }
 fprintf(fp, "Closing the UART file.\n");
 fclose (fp);
 }
 return 0;
}
```

Related Information

Nios II Software Developer's Handbook

11.4.1.1. Driver Options: Fast vs Small Implementations

To accommodate the requirements of different types of systems, the UART driver provides two variants: a fast version and a small version. The fast version is the default. Both fast and small drivers fully support the C standard library functions and the HAL API.

The fast driver is an interrupt-driven implementation, which allows the processor to perform other tasks when the device is not ready to send or receive data. Because the UART data rate is slow compared to the processor, the fast driver can provide a large performance benefit for systems that could be performing other tasks in the interim.

The small driver is a polled implementation that waits for the UART hardware before sending and receiving each character. There are two ways to enable the small footprint driver:

- Enable the small footprint setting for the HAL system library project. This option affects device drivers for all devices in the system as well.
- Specify the preprocessor option `-DALTERA_AVALON_UART_SMALL`. You can use this option if you want the small, polled implementation of the UART driver, but do not want to affect the drivers for other devices.

Refer to the help system in the Nios II IDE for details about how to set HAL properties and preprocessor options.

11.4.1.1.1. UART API

Table 88. altera_avalon_uart_close

`altera_avalon_uart_close` is not available for small driver.

Prototype	`int altera_avalon_uart_close(altera_avalon_uart_state* sp, int flags);`
Include	`<altera_avalon_uart.h>`
Parameters	• sp - the UART device to close • flags - for indicating blocking/non-blocking access for single/multi-threaded
Returns	None
Description	Closes UART device

Table 89. altera_avalon_uart_read

Prototype	`int altera_avalon_uart_read(altera_avalon_uart_state* sp, char* ptr, int len, int flags);`
Include	`<altera_avalon_uart.h>`
Parameters	• sp - the UART device • ptr - destination address • len - maximum length of the data • flags - for indicating blocking/non-blocking access for single/multi-threaded
Returns	Number of bytes read
Description	Reads data to the UART receiver buffer

Table 90. altera_avalon_uart_write

Prototype	`int altera_avalon_uart_write(altera_avalon_uart_state* sp, char* ptr, int len, int flags);`
Include	`<altera_avalon_uart.h>`
Parameters	• sp - the UART device • ptr - source address • len - maximum length of the data • flags - for indicating blocking/non-blocking access for single/multi-threaded
Returns	Number of bytes written
Description	Writes data to the UART receiver buffer

Table 91. altera_avalon_uart_irq

`altera_avalon_uart_irq` is not available for small driver.

Prototype	`static void altera_avalon_uart_irq(void* context)`
Include	`<altera_avalon_uart.h>`
Parameters	• context - the UART device
Returns	None
Description	Interrupt handler to process UART interrupts to process receiver/transmitter interrupts

Table 92. altera_avalon_uart_rxirq

`altera_avalon_uart_rxirq` is not available for small driver.

Prototype	`static void altera_avalon_uart_rxirq(altera_avalon_uart_state* sp, alt_u32 status)`
Include	`<altera_avalon_uart.h>`
Parameters	• sp - the UART device • status - individual bits that indicate conditions inside the UART core
Returns	None
Description	Process a receive interrupt. It transfers the incoming character into the receiver circular buffer and sets the appropriate flags to indicate that there is data ready to be processed

Table 93. altera_avalon_uart_txirq

`altera_avalon_uart_txirq` is not available for small driver.

Prototype	`static void altera_avalon_uart_txirq(altera_avalon_uart_state* sp, alt_u32 status)`
Include	`<altera_avalon_uart.h>`
Parameters	• sp - the UART device • status - individual bits that indicate conditions inside the UART core
Returns	None
Description	Process a transmit interrupt. It transfers data from the transmit buffer to the device and sets the appropriate flags to indicate that there is data ready to be processed

11.4.1.1.2. UART Device Structure

```
typedef struct altera_avalon_uart_state_s
{
  void*            base;            /* The base address of the device */
  alt_u32          ctrl;            /* Shadow value of the control register */
  volatile alt_u32 rx_start;        /* Start of the pending receive data */
  volatile alt_u32 rx_end;          /* End of the pending receive data */
  volatile alt_u32 tx_start;        /* Start of the pending transmit data */
  volatile alt_u32 tx_end;          /* End of the pending transmit data */
#ifdef ALTERA_AVALON_UART_USE_IOCTL
  struct termios termios;           /* Current device configuration */
  alt_u32          freq;            /* Current baud rate */
#endif
  alt_u32          flags;           /* Configuration flags */
  ALT_FLAG_GRP     (events)         /* Event flags used for
                                     * foreground/background in multi-threaded
                                     * mode */
  ALT_SEM          (read_lock)      /* Semaphore used to control access to the
                                     * read buffer in multi-threaded mode */
  ALT_SEM          (write_lock)     /* Semaphore used to control access to the
                                     * write buffer in multi-threaded mode */
  volatile alt_u8  rx_buf[ALT_AVALON_UART_BUF_LEN]; /* The receive buffer */
  volatile alt_u8  tx_buf[ALT_AVALON_UART_BUF_LEN]; /* The transmit buffer */
} altera_avalon_uart_state;
```

11.4.1.2. `ioctl()` Operations

The UART driver supports the `ioctl()` function to allow HAL-based programs to request device-specific operations. The table below defines operation requests that the UART driver supports.

Table 94. UART ioctl() Operations

Request	Description
`TIOCEXCL`	Locks the device for exclusive access. Further calls to `open()` for this device will fail until either this file descriptor is closed, or the lock is released using the `TIOCNXCL ioctl` request. For this request to succeed there can be no other existing file descriptors for this device. The parameter `arg` is ignored.
`TIOCNXCL`	Releases a previous exclusive access lock. The parameter `arg` is ignored.

Additional operation requests are also optionally available for the fast driver only, as shown in **Optional UART ioctl() Operations for the Fast Driver Only** Table. To enable these operations in your program, you must set the preprocessor option `-DALTERA_AVALON_UART_USE_IOCTL`.

Table 95. Optional UART ioctl() Operations for the Fast Driver Only

Request	Description
`TIOCMGET`	Returns the current configuration of the device by filling in the contents of the input termios structure. A pointer to this structure is supplied as the value of the parameter `opt`.
`TIOCMSET`	Sets the configuration of the device according to the values contained in the input termios structure. A pointer to this structure is supplied as the value of the parameter `arg`.

Note: The termios structure is defined by the Newlib C standard library. You can find the definition in the file <Nios II EDS install path>`/components/altera_hal/HAL/inc/sys/termios.h`.

For details about the `ioctl()` function, refer to the *Nios II Software Developer's Handbook* .

Related Information

Nios II Software Developer's Handbook

11.4.1.3. Limitations

The HAL driver for the UART core does not support the endofpacket register. Refer to the Register map section for details.

11.4.2. Software Files

The UART core is accompanied by the following software files. These files define the low-level interface to the hardware, and provide the HAL drivers. Application developers should not modify these files.

- `altera_avalon_uart_regs.h`—This file defines the core's register map, providing symbolic constants to access the low-level hardware. The symbols in this file are used only by device driver functions.
- `altera_avalon_uart.h`, `altera_avalon_uart_fd.c`, `altera_avalon_uart_init.c`, `altera_avalon_uart_ioctl.c`, `altera_avalon_uart_read.c`, `altera_avalon_uart_write.c`—These files implement the UART core device driver for the HAL system library.

11.4.3. Register Map

Programmers using the HAL API never access the UART core directly via its registers. In general, the register map is only useful to programmers writing a device driver for the core.

The Intel-provided HAL device driver accesses the device registers directly. If you are writing a device driver and the HAL driver is active for the same device, your driver will conflict and fail to operate.

The **UART Core Register map** table below shows the register map for the UART core. Device drivers control and communicate with the core through the memory-mapped registers.

Table 96. UART Core Register Map

Offset	Register Name	R/W	Description/Register Bits													
			15:13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	rxdata	RO	Reserved					(16)	(16)	Receive Data						
1	txdata	WO	Reserved					(16)	(16)	Transmit Data						
2	status (15)	RW	Reserved	eop	cts	dcts		e	rrdy	trdy	tmt	toe	roe	brk	fe	pe
3	control	RW	Reserved	ieop	rts	idcts	trbk	ie	irrdy	itrdy	itmt	itoe	iroe	ibrk	ife	ipe
4	divisor (17)	RW	Baud Rate Divisor													
5	endof-packet (17)	RW	Reserved					(16)	(16)	End-of-Packet Value						

Some registers and bits are optional. These registers and bits exists in hardware only if it was enabled at system generation time. Optional registers and bits are noted in the following sections.

11.4.3.1. rxdata Register

The `rxdata` register holds data received via the RXD input. When a new character is fully received via the RXD input, it is transferred into the `rxdata` register, and the `status` register's `rrdy` bit is set to 1. The `status` register's `rrdy` bit is set to 0 when the `rxdata` register is read. If a character is transferred into the `rxdata` register while the `rrdy` bit is already set (in other words, the previous character was not retrieved), a receiver-overrun error occurs and the `status` register's `roe` bit is set to 1. New characters are always transferred into the `rxdata` register, regardless of whether the previous character was read. Writing data to the `rxdata` register has no effect.

11.4.3.2. txdata Register

Avalon-MM host peripherals write characters to be transmitted into the `txdata` register. Characters should not be written to `txdata` until the transmitter is ready for a new character, as indicated by the TRDY bit in the `status` register. The TRDY bit is set to 0 when a character is written into the `txdata` register. The TRDY bit is set to 1

(15) Writing zero to the `status` register clears the `dcts`, `e`, `toe`, `roe`, `brk`, `fe`, and `pe` bits.

(16) These bits may or may not exist, depending on the **Data Width** hardware option. If they do not exist, they read zero, and writing has no effect.

(17) This register may or may not exist, depending on hardware configuration options. If it does not exist, reading returns an undefined value and writing has no effect.

when the character is transferred from the txdata register into the transmitter shift register. If a character is written to the txdata register when TRDY is 0, the result is undefined. Reading the txdata register returns an undefined value.

For example, assume the transmitter logic is idle and an Avalon-MM host peripheral writes a first character into the txdata register. The TRDY bit is set to 0, then set to 1 when the character is transferred into the transmitter shift register. The host can then write a second character into the txdata register, and the TRDY bit is set to 0 again. However, this time the shift register is still busy shifting out the first character to the TXD output. The TRDY bit is not set to 1 until the first character is fully shifted out and the second character is automatically transferred into the transmitter shift register.

11.4.3.3. status Register

The status register consists of individual bits that indicate particular conditions inside the UART core. Each status bit is associated with a corresponding interrupt-enable bit in the control register. The status register can be read at any time. Reading does not change the value of any of the bits. Writing zero to the status register clears the DCTS, E, TOE, ROE, BRK, FE, and PE bits.

Table 97. status Register Bits

Bit	Name	Access	Description
0 (18)	PE	RC	Parity error. A parity error occurs when the received parity bit has an unexpected (incorrect) logic level. The PE bit is set to 1 when the core receives a character with an incorrect parity bit. The PE bit stays set to 1 until it is explicitly cleared by a write to the status register. When the PE bit is set, reading from the rxdata register produces an undefined value. If the **Parity** hardware option is not enabled, no parity checking is performed and the PE bit always reads 0. Refer to **Data Bits, Stop, Bits, Parity** section.
1	FE	RC	Framing error. A framing error occurs when the receiver fails to detect a correct stop bit. The FE bit is set to 1 when the core receives a character with an incorrect stop bit. The FE bit stays set to 1 until it is explicitly cleared by a write to the status register. When the FE bit is set, reading from the rxdata register produces an undefined value.
2	BRK	RC	Break detect. The receiver logic detects a break when the RXD pin is held low (logic 0) continuously for longer than a full-character time (data bits, plus start, stop, and parity bits). When a break is detected, the BRK bit is set to 1. The BRK bit stays set to 1 until it is explicitly cleared by a write to the status register.
3	ROE	RC	Receive overrun error. A receive-overrun error occurs when a newly received character is transferred into the rxdata holding register before the previous character is read (in other words, while the RRDY bit is 1). In this case, the ROE bit is set to 1, and the previous contents of rxdata are overwritten with the new character. The ROE bit stays set to 1 until it is explicitly cleared by a write to the status register.
4	TOE	RC	Transmit overrun error. A transmit-overrun error occurs when a new character is written to the txdata holding register before the previous character is transferred into the shift register (in other words, while the TRDY bit is 0). In this case the TOE bit is set to 1. The TOE bit stays set to 1 until it is explicitly cleared by a write to the status register.
5	TMT	R	Transmit empty. The TMT bit indicates the transmitter shift register's current state. When the shift register is in the process of shifting a character out the TXD pin, TMT is set to 0. When the shift register is idle (in other words, a

continued...

Bit	Name	Access	Description
			character is not being transmitted) the TMT bit is 1. An Avalon-MM host peripheral can determine if a transmission is completed (and received at the other end of a serial link) by checking the TMT bit.
6	TRDY	R	Transmit ready. The TRDY bit indicates the `txdata` holding register's current state. When the `txdata` register is empty, it is ready for a new character, and TRDY is 1. When the `txdata` register is full, TRDY is 0. An Avalon-MM host peripheral must wait for TRDY to be 1 before writing new data to `txdata`.
7	RRDY	R	Receive character ready. The RRDY bit indicates the `rxdata` holding register's current state. When the `rxdata` register is empty, it is not ready to be read and RRDY is 0. When a newly received value is transferred into the `rxdata` register, RRDY is set to 1. Reading the `rxdata` register clears the RRDY bit to 0. An Avalon-MM host peripheral must wait for RRDY to equal 1 before reading the `rxdata` register.
8	E	RC	Exception. The E bit indicates that an exception condition occurred. The E bit is a logical-OR of the TOE, ROE, BRK, FE, and PE bits. The E bit and its corresponding interrupt-enable bit (IE) bit in the `control` register provide a convenient method to enable/disable IRQs for all error conditions. The E bit is set to 0 by a write operation to the `status` register.
10 (18)	DCTS	RC	Change in clear to send (CTS) signal. The DCTS bit is set to 1 whenever a logic-level transition is detected on the CTS_N input port (sampled synchronously to the Avalon-MM clock). This bit is set by both falling and rising transitions on CTS_N. The DCTS bit stays set to 1 until it is explicitly cleared by a write to the `status` register.
11 (18)	CTS	R	Clear-to-send (CTS) signal. The CTS bit reflects the CTS_N input's instantaneous state (sampled synchronously to the Avalon-MM clock). The CTS_N input has no effect on the transmit or receive processes. The only visible effect of the CTS_N input is the state of the CTS and DCTS bits, and an IRQ that can be generated when the control register's idcts bit is enabled.
12 (18)	EOP	R(18)	End of packet encountered. The EOP bit is set to 1 by one of the following events: An EOP character is written to `txdata` An EOP character is read from `rxdata` The EOP character is determined by the contents of the `endofpacket` register. The EOP bit stays set to 1 until it is explicitly cleared by a `write` to the `status` register. If the **Include End-of-Packet Register** hardware option is not enabled, the EOP bit always reads 0. Refer to Streaming Data (DMA) Control Section.

11.4.3.4. control Register

The `control` register consists of individual bits, each controlling an aspect of the UART core's operation. The value in the `control` register can be read at any time.

Each bit in the `control` register enables an IRQ for a corresponding bit in the `status` register. When both a status bit and its corresponding interrupt-enable bit are 1, the core generates an IRQ.

(18) This bit is optional and may not exist in hardware.

Table 98. control Register Bits

Bit	Name	Access	Description
0	IPE	RW	Enable interrupt for a parity error.
1	IFE	RW	Enable interrupt for a framing error.
2	IBRK	RW	Enable interrupt for a break detect.
3	IROE	RW	Enable interrupt for a receiver overrun error.
4	ITOE	RW	Enable interrupt for a transmitter overrun error.
5	ITMT	RW	Enable interrupt for a transmitter shift register empty.
6	ITRDY	RW	Enable interrupt for a transmission ready.
7	IRRDY	RW	Enable interrupt for a read ready.
8	IE	RW	Enable interrupt for an exception.
9	TRBK	RW	Transmit break. The TRBK bit allows an Avalon-MM host peripheral to transmit a break character over the TXD output. The TXD signal is forced to 0 when the TRBK bit is set to 1. The TRBK bit overrides any logic level that the transmitter logic would otherwise drive on the TXD output. The TRBK bit interferes with any transmission in process. The Avalon-MM host peripheral must set the TRBK bit back to 0 after an appropriate break period elapses.
10[19]	IDCTS	RW	Enable interrupt for a change in CTS signal.
11[19]	RTS	RW	Request to send (RTS) signal. The RTS bit directly feeds the RTS_N output. An Avalon-MM host peripheral can write the RTS bit at any time. The value of the RTS bit only affects the RTS_N output; it has no effect on the transmitter or receiver logic. Because the RTS_N output is logic negative, when the RTS bit is 1, a low logic-level (0) is driven on the RTS_N output.
12[19]	IEOP	RW	Enable interrupt for end-of-packet condition.

11.4.3.5. divisor Register (Optional)

The value in the `divisor` register is used to generate the baud rate clock. The effective baud rate is determined by the formula:

Baud Rate = (Clock frequency) / (divisor + 1)

The `divisor` register is an optional hardware feature. If the **Baud Rate Can Be Changed By Software** hardware option is not enabled, the `divisor` register does not exist. In this case, writing `divisor` has no effect, and reading `divisor` returns an undefined value. For more information, refer to the **Baud Rate Options** section.

11.4.3.6. endofpacket Register (Optional)

The value in the `endofpacket` register determines the end-of-packet character for variable-length DMA transactions. After reset, the default value is zero, which is the ASCII null character (\0). For more information, refer to **status Register bits** for the description for the EOP bit.

(19) This bit is optional and may not exist in hardware.

The endofpacket register is an optional hardware feature. If the **Include end-of-packet register** hardware option is not enabled, the endofpacket register does not exist. In this case, writing endofpacket has no effect, and reading returns an undefined value.

11.4.4. Interrupt Behavior

The UART core outputs a single IRQ signal to the Avalon-MM interface, which can connect to any host peripheral in the system, such as a Nios II processor. The host peripheral must read the status register to determine the cause of the interrupt.

Every interrupt condition has an associated bit in the status register and an interrupt-enable bit in the control register. When any of the interrupt conditions occur, the associated status bit is set to 1 and remains set until it is explicitly acknowledged. The IRQ output is asserted when any of the status bits are set while the corresponding interrupt-enable bit is 1. A host peripheral can acknowledge the IRQ by clearing the status register.

At reset, all interrupt-enable bits are set to 0; therefore, the core cannot assert an IRQ until a host peripheral sets one or more of the interrupt-enable bits to 1.

All possible interrupt conditions are listed with their associated status and control (interrupt-enable) bits. Details of each interrupt condition are provided in the status bit descriptions.

11.5. UART Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2020.01.22	19.4	Corrected *Table: control Register Bits*,
2018.09.24	18.1	• Added two new sections: — *UART API* — *UART Device Structure* • Corrected the divisor and baud rate formula in *Baud Rate (bps) Setting* section.
2018.05.07	18.0	• Corrected the *Software Files* that implement the UART core device driver for the HAL system library • Removed the following sections: — Simulation Settings — Simulation Consideration • Fixed typos and references.

Date and Document Version	Version	Changes
June 2016	2016.06.17	Removed content regarding Avalon-MM flow control.
December 2010	v10.1.0	Removed the "Device Support", "Instantiating the Core in SOPC Builder", and "Referenced Documents" sections.
July 2010	v10.0.0	No change from previous release.
November 2009	v9.1.0	No change from previous release.

continued...

Date and Document Version	Version	Changes
March 2009	v9.0.0	Added description of a new parameter, **Synchronizer stages**.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	v8.0.0	No change from previous release.

12. JTAG UART Core

12.1. Core Overview

The JTAG UART core with Avalon interface implements a method to communicate serial character streams between a host PC and a Platform Designer system on an Intel FPGA. In many designs, the JTAG UART core eliminates the need for a separate RS-232 serial connection to a host PC for character I/O. The core provides an Avalon interface that hides the complexities of the JTAG interface from embedded software programmers. Host peripherals (such as a Nios II processor) communicate with the core by reading and writing control and data registers.

The JTAG UART core uses the JTAG circuitry built in to Intel FPGAs, and provides host access via the JTAG pins on the FPGA. The host PC can connect to the FPGA via any Intel FPGA JTAG download cable, such as the Intel FPGA download cable II. Software support for the JTAG UART core is provided by Intel. For the Nios II processor, device drivers are provided in the hardware abstraction layer (HAL) system library, allowing software to access the core using the ANSI C Standard Library `stdio.h` routines.

Nios II processor users can access the JTAG UART via the Nios II IDE or the **nios2-terminal** command-line utility. For further details, refer to the Nios II Software Developer's Handbook or the Nios II IDE online help.

For the host PC, Intel provides JTAG terminal software that manages the connection to the target, decodes the JTAG data stream, and displays characters on screen.

The JTAG UART core is Platform Designer-ready and integrates easily into any Platform Designer-generated system.

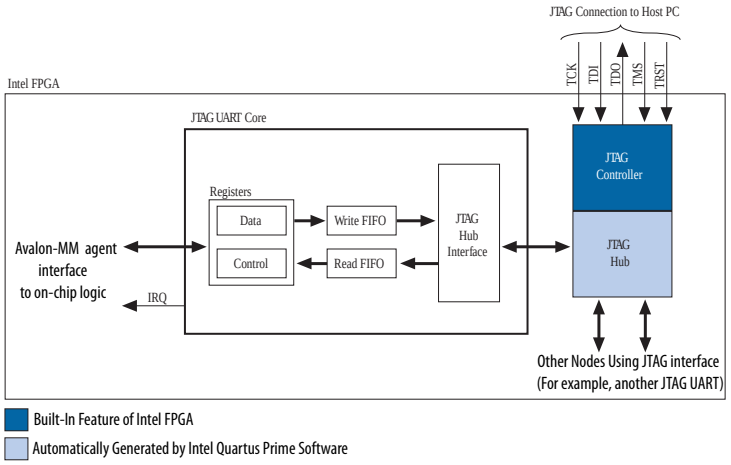
12.2. Functional Description

The figure below shows a block diagram of the JTAG UART core and its connection to the JTAG circuitry inside an Intel FPGA. The following sections describe the components of the core.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Figure 40. JTAG UART Core Block Diagram



12.2.1. Avalon Agent Interface and Registers

The JTAG UART core provides an Avalon agent interface to the JTAG circuitry on an Intel FPGA. The user-visible interface to the JTAG UART core consists of two 32-bit registers, `data` and `control`, that are accessed through an Avalon agent port. An Avalon host, such as a Nios II processor, accesses the registers to control the core and transfer data over the JTAG connection. The core operates on 8-bit units of data at a time; eight bits of the `data` register serve as a one-character payload.

The JTAG UART core provides an active-high interrupt output that can request an interrupt when read data is available, or when the write FIFO is ready for data. For further details see the **Interrupt Behavior** section.

12.2.2. Read and Write FIFOs

The JTAG UART core provides bidirectional FIFOs to improve bandwidth over the JTAG connection. The FIFO depth is parameterizable to accommodate the available on-chip memory. The FIFOs can be constructed out of memory blocks or registers, allowing you to trade off logic resources for memory resources, if necessary.

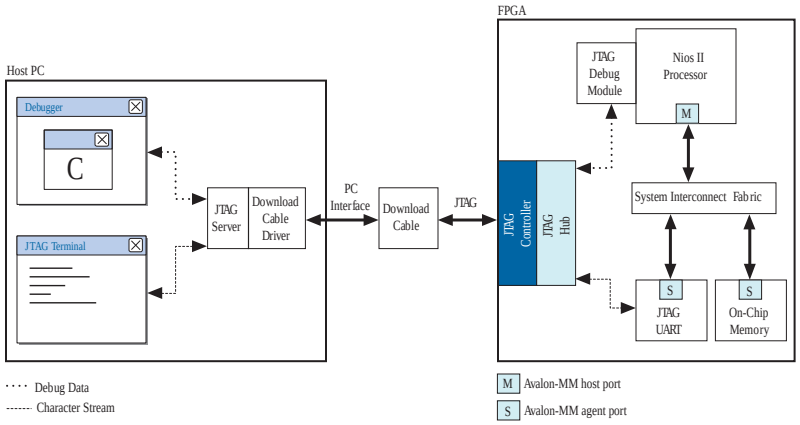
12.2.3. JTAG Interface

Intel FPGAs contain built-in JTAG control circuitry between the device's JTAG pins and the logic inside the device. The JTAG controller can connect to user-defined circuits called nodes implemented in the FPGA. Because several nodes may need to communicate via the JTAG interface, a JTAG hub, which is a multiplexer, is necessary. During logic synthesis and fitting, the Intel Quartus Prime software automatically generates the JTAG hub logic. No manual design effort is required to connect the JTAG circuitry inside the device; the process is presented here only for clarity.

12.2.4. Host-Target Connection

Below you can see the connection between a host PC and an Platform Designer-generated system containing a JTAG UART core.

Figure 41. Example System Using the JTAG UART Core



The JTAG controller on the FPGA and the download cable driver on the host PC implement a simple data-link layer between host and target. All JTAG nodes inside the FPGA are multiplexed through the single JTAG connection. JTAG server software on the host PC controls and decodes the JTAG data stream, and maintains distinct connections with nodes inside the FPGA.

The example system in the figure above contains one JTAG UART core and a Nios II processor. Both agents communicate with the host PC over a single Intel FPGA download cable. Thanks to the JTAG server software, each host application has an independent connection to the target. Intel provides the JTAG server drivers and host software required to communicate with the JTAG UART core.

Systems with multiple JTAG UART cores are possible, and all cores communicate via the same JTAG interface. To maintain coherent data streams, only one processor should communicate with each JTAG UART core.

12.3. Configuration

The following sections describe the available configuration options.

12.3.1. Configuration Page

The options on this page control the hardware configuration of the JTAG UART core. The default settings are pre-configured to behave optimally with the Intel-provided device drivers and JTAG terminal software. Most designers should not change the default values, except for the **Construct using registers instead of memory blocks** option.

12.3.1.1. Write FIFO Settings

The write FIFO buffers data flowing from the Avalon interface to the host. The following settings are available:

- **Depth**—The write FIFO depth can be set from 8 to 32,768 bytes. Only powers of two are allowed. Larger values consume more on-chip memory resources. A depth of 64 is generally optimal for performance, and larger values are rarely necessary.
- **IRQ Threshold—**The write IRQ threshold governs how the core asserts its IRQ in response to the FIFO emptying. As the JTAG circuitry empties data from the write FIFO, the core asserts its IRQ when the number of characters remaining in the FIFO reaches this threshold value. For maximum bandwidth, a processor should service the interrupt by writing more data and preventing the write FIFO from emptying completely. A value of 8 is typically optimal. See the **Interrupt Behavior** section for further details.
- **Construct using registers instead of memory blocks**—Turning on this option causes the FIFO to be constructed out of on-chip logic resources. This option is useful when memory resources are limited. Each byte consumes roughly 11 logic elements (LEs), so a FIFO depth of 8 (bytes) consumes roughly 88 LEs.

12.3.1.2. Read FIFO Settings

The read FIFO buffers data flowing from the host to the Avalon interface. Settings are available to control the depth of the FIFO and the generation of interrupts.

- **Depth**—The read FIFO depth can be set from 8 to 32,768 bytes. Only powers of two are allowed. Larger values consume more on-chip memory resources. A depth of 64 is generally optimal for performance, and larger values are rarely necessary.
- **IRQ Threshold—**The IRQ threshold governs how the core asserts its IRQ in response to the FIFO filling up. As the JTAG circuitry fills up the read FIFO, the core asserts its IRQ when the amount of space remaining in the FIFO reaches this threshold value. For maximum bandwidth, a processor should service the interrupt by reading data and preventing the read FIFO from filling up completely. A value of 8 is typically optimal. See the **Interrupt Behavior** section for further details.
- **Construct using registers instead of memory blocks**—Turning on this option causes the FIFO to be constructed out of logic resources. This option is useful when memory resources are limited. Each byte consumes roughly 11 LEs, so a FIFO depth of 8 (bytes) consumes roughly 88 LEs.

12.4. Software Programming Model

The following sections describe the software programming model for the JTAG UART core, including the register map and software declarations to access the hardware. For Nios II processor users, Intel provides HAL system library drivers that enable you to access the JTAG UART using the ANSI C standard library functions, such as `printf()` and `getchar()`.

12.4.1. HAL System Library Support

The Intel-provided driver implements a HAL character-mode device driver that integrates into the HAL system library for Nios II systems. HAL users should access the JTAG UART via the familiar HAL API and the ANSI C standard library, rather than accessing the JTAG UART registers. `ioctl()` requests are defined that allow HAL users to control the hardware-dependent aspects of the JTAG UART.

Note: If your program uses the Intel-provided HAL device driver to access the JTAG UART hardware, accessing the device registers directly will interfere with the correct behavior of the driver.

For Nios II processor users, the HAL system library API provides complete access to the JTAG UART core's features. Nios II programs treat the JTAG UART core as a character mode device, and send and receive data using the ANSI C standard library functions, such as `getchar()` and `printf()`.

The "Printing Characters to a JTAG UART core as stdout" example below demonstrates the simplest possible usage, printing a message to `stdout` using `printf()`. In this example, the Platform Designer system contains a JTAG UART core, and the HAL system library is configured to use this JTAG UART device for `stdout`.

Table 99. Example: Printing Characters to a JTAG UART Core as stdout

```
#include <stdio.h>
int main ()
{
printf("Hello world.\n");
return 0;
}
```

The **Transmitting characters to a JTAG UART Core** example demonstrates reading characters from and sending messages to a JTAG UART core using the C standard library. In this example, the Platform Designer system contains a JTAG UART core named `jtag_uart` that is not necessarily configured as the `stdout` device. In this case, the program treats the device like any other node in the HAL file system.

Table 100. Example: Transmitting Characters to a JTAG UART Core

```
/* A simple program that recognizes the characters 't' and 'v' */
#include <stdio.h>
#include <string.h>
int main ()
{
char* msg = "Detected the character 't'.\n";
FILE* fp;
char prompt = 0;
fp = fopen ("/dev/jtag_uart", "r+"); //Open file for reading and writing
if (fp)
{
while (prompt != 'v')
{ // Loop until we receive a 'v'.
prompt = getc(fp); // Get a character from the JTAG UART.
if (prompt == 't')
{ // Print a message if character is 't'.
fwrite (msg, strlen (msg), 1, fp);
}
if (ferror(fp)) // Check if an error occurred with the file
clearerr(fp); // If so, clear it.
}
fprintf(fp, "Closing the JTAG UART file handle.\n");
fclose (fp);
}
return 0;
}
```

In this example, the `ferror(fp)` is used to check if an error occurred on the JTAG UART connection, such as a disconnected JTAG connection. In this case, the driver detects that the JTAG connection is disconnected, reports an error (`EIO`), and discards data for subsequent transactions. If this error ever occurs, the C library latches the value until you explicitly clear it with the `clearerr()` function.

For complete details of the HAL system library, refer to the*Nios II Software Developer's Handbook* .

The Nios II Embedded Design Suite (EDS) provides a number of software example designs that use the JTAG UART core.

Related Information

Nios II Software Developer's Handbook

12.4.1.1. Driver Options: Fast vs. Small Implementations

To accommodate the requirements of different types of systems, the JTAG UART driver has two variants, a fast version and a small version. The fast behavior is used by default. Both the fast and small drivers fully support the C standard library functions and the HAL API.

The fast driver is an interrupt-driven implementation, which allows the processor to perform other tasks when the device is not ready to send or receive data. Because the JTAG UART data rate is slow compared to the processor, the fast driver can provide a large performance benefit for systems that could be performing other tasks in the interim. In addition, the fast version of the Intel FPGA Avalon JTAG UART monitors the connection to the host. The driver discards characters if no host is connected, or if the host is not running an application that handles the I/O stream.

The small driver is a polled implementation that waits for the JTAG UART hardware before sending and receiving each character. The performance of the small driver is poor if you are sending large amounts of data. The small version assumes that the host is always connected, and will never discard characters. Therefore, the small driver will hang the system if the JTAG UART hardware is ever disconnected from the host while the program is sending or receiving data. There are two ways to enable the small footprint driver:

- Enable the small footprint setting for the HAL system library project. This option affects device drivers for all devices in the system.
- Specify the preprocessor option `-DALTERA_AVALON_JTAG_UART_SMALL`. Use this option if you want the small, polled implementation of the JTAG UART driver, but you do not want to affect the drivers for other devices.

12.4.1.2. ioctl() Operations

The fast version of the JTAG UART driver supports the `ioctl()` function to allow HAL-based programs to request device-specific operations. Specifically, you can use the `ioctl()` operations to control the timeout period, and to detect whether or not a host is connected. The fast driver defines the `ioctl()` operations shown in below.

Table 101. JTAG UART ioctl() Operations for the Fast Driver Only

Request	Meaning
TIOCSTIMEOUT	Set the timeout (in seconds) after which the driver will decide that the host is not connected. A timeout of 0 makes the target assume that the host is always connected. The `ioctl` arg parameter passed in must be a pointer to an integer.
TIOCGCONNECTED	Sets the integer arg parameter to a value that indicates whether the host is connected and acting as a terminal (1), or not connected (0). The `ioctl` arg parameter passed in must be a pointer to an integer.

For details about the `ioctl()` function, refer to the*Nios II Software Developer's Handbook* .

Related Information

Nios II Software Developer's Handbook

12.4.2. Software Files

The JTAG UART core is accompanied by the following software files. These files define the low-level interface to the hardware, and provide the HAL drivers. Application developers should not modify these files.

- `altera_avalon_jtag_uart_regs.h`—This file defines the core's register map, providing symbolic constants to access the low-level hardware. The symbols in this file are used only by device driver functions.
- `altera_avalon_jtag_uart.h`, `altera_avalon_jtag_uart_init.c`, `altera_avalon_jtag_uart_fd.c`, `altera_avalon_jtag_uart_ioctl.c`, `altera_avalon_jtag_uart_read.c`, `altera_avalon_jtag_uart_write.c` —These files implement the HAL system library device driver.

12.4.3. Accessing the JTAG UART Core via a Host PC

Host software is necessary for a PC to access the JTAG UART core. The Nios II IDE supports the JTAG UART core, and displays character I/O in a console window. Intel also provides a command-line utility called **nios2-terminal** that opens a terminal session with the JTAG UART core.

For further details, refer to the *Nios II Software Developer's Handbook* and Nios II IDE online help.

Related Information

Nios II Software Developer's Handbook

12.4.4. Register Map

Programmers using the HAL API never access the JTAG UART core directly via its registers. In general, the register map is only useful to programmers writing a device driver for the core.

Note: The Intel-provided HAL device driver accesses the device registers directly. If you are writing a device driver, and the HAL driver is active for the same device, your driver will conflict and fail to operate.

The table below shows the register map for the JTAG UART core. Device drivers control and communicate with the core through the two, 32-bit memory-mapped registers.

Table 102. JTAG UART Core Register Map

Offset	Register Name	R/W	Bit Description													
			31	..	16	15	14	..	11	10	9	8	7	..	2	1	0
0	data	RW	RAVAIL			RVALID	Reserved						DATA				
1	control	RW	WSPACE			Reserved				AC	WI	RI	Reserved			WE	RE

Note: Reserved fields—Read values are undefined. Write zero.

12.4.4.1. Data Register

Embedded software accesses the read and write FIFOs via the `data` register. The table below describes the function of each bit.

Table 103. data Register Bits

Bit(s)	Name	Access	Description
[7:0]	DATA	R/W	The value to transfer to/from the JTAG core. When writing, the DATA field holds a character to be written to the write FIFO. When reading, the DATA field holds a character read from the read FIFO.
[15]	RVALID	R	Indicates whether the DATA field is valid. If RVALID=1, the DATA field is valid, otherwise DATA is undefined.
[32:16]	RAVAIL	R	The number of characters remaining in the read FIFO (after the current read).

A read from the `data` register returns the first character from the FIFO (if one is available) in the DATA field. Reading also returns information about the number of characters remaining in the FIFO in the RAVAIL field. A write to the `data` register stores the value of the DATA field in the write FIFO. If the write FIFO is full, the character is lost.

12.4.4.2. Control Register

Embedded software controls the JTAG UART core's interrupt generation and reads status information via the `control` register. The Control Register Bits table below describes the function of each bit.

Table 104. Control Register Bits

Bit(s)	Name	Access	Description
0	RE	R/W	Interrupt-enable bit for read interrupts.
1	WE	R/W	Interrupt-enable bit for write interrupts.
8	RI	R	Indicates that the read interrupt is pending.

continued...

Bit(s)	Name	Access	Description
9	WI	R	Indicates that the write interrupt is pending.
10	AC	R/C	Indicates that there has been JTAG activity since the bit was cleared. Writing 1 to AC clears it to 0.
[32:16]	WSPACE	R	The number of spaces available in the write FIFO.

A read from the `control` register returns the status of the read and write FIFOs. Writes to the register can be used to enable/disable interrupts, or clear the AC bit.

The RE and WE bits enable interrupts for the read and write FIFOs, respectively. The WI and RI bits indicate the status of the interrupt sources, qualified by the values of the interrupt enable bits (WE and RE). Embedded software can examine RI and WI to determine the condition that generated the IRQ. See the **Interrupt Behavior** section for further details.

The AC bit indicates that an application on the host PC has polled the JTAG UART core via the JTAG interface. Once set, the AC bit remains set until it is explicitly cleared via the Avalon interface. Writing 1 to AC clears it. Embedded software can examine the AC bit to determine if a connection exists to a host PC. If no connection exists, the software may choose to ignore the JTAG data stream. When the host PC has no data to transfer, it can choose to poll the JTAG UART core as infrequently as once per second. Delays caused by other host software using the JTAG download cable could cause delays of up to 10 seconds between polls.

12.4.5. Interrupt Behavior

The JTAG UART core generates an interrupt when either of the individual interrupt conditions is pending and enabled.

Interrupt behavior is of interest to device driver programmers concerned with the bandwidth performance to the host PC. Example designs and the JTAG terminal program provided with Nios II Embedded Design Suite (EDS) are pre-configured with optimal interrupt behavior.

The JTAG UART core has two kinds of interrupts: write interrupts and read interrupts. The WE and RE bits in the `control` register enable/disable the interrupts.

The core can assert a write interrupt whenever the write FIFO is nearly empty. The nearly empty threshold, `write_threshold`, is specified at system generation time and cannot be changed by embedded software. The write interrupt condition is set whenever there are `write_threshold` or fewer characters in the write FIFO. It is cleared by writing characters to fill the write FIFO beyond the `write_threshold`. Embedded software should only enable write interrupts after filling the write FIFO. If it has no characters remaining to send, embedded software should disable the write interrupt.

The core can assert a read interrupt whenever the read FIFO is nearly full. The nearly full threshold value, `read_threshold`, is specified at system generation time and cannot be changed by embedded software. The read interrupt condition is set whenever the read FIFO has `read_threshold` or fewer spaces remaining. The read interrupt condition is also set if there is at least one character in the read FIFO and no more characters are expected. The read interrupt is cleared by reading characters from the read FIFO.

For optimum performance, the interrupt thresholds should match the interrupt response time of the embedded software. For example, with a 10-MHz JTAG clock, a new character is provided (or consumed) by the host PC every 1 µs. With a threshold of 8, the interrupt response time must be less than 8 µs. If the interrupt response time is too long, performance suffers. If it is too short, interrupts occurs too often.

For Nios II processor systems, read and write thresholds of 8 are an appropriate default.

12.5. JTAG UART Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0	• Removed the following sections: — Simulation Settings — Hardware Simulation Considerations • Implemented editorial enhancements.

Date	Version	Changes
July 2014	2014.07.24	Removed mention of SOPC Builder, updated to Platform Designer
December 2010	v10.1.0	Removed the "Device Support", "Instantiating the Core in SOPC Builder", and "Referenced Documents" sections.
July 2010	v10.0.0	No change from previous release.
November 2009	v9.1.0	No change from previous release.
March 2009	v9.0.0	No change from previous release.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	v8.0.0	No change from previous release.

13. Intel FPGA Avalon Mailbox Core

13.1. Core Overview

In a multiprocessor design, each processor may be dedicated to perform a specific task. Communication between processors becomes crucial if the tasks of each individual processor are interdependent. Communication between processors may involve data passing or task control coordination to accomplish certain functions.

The Intel FPGA Avalon Mailbox component provides the medium of communication between processors. It provides a "message" passing location between the sending processor and receiving processor. The receiving processor is notified upon a message arrival. The notification can be in the form of an interrupt issuing to the receiving processor or it can continue polling for new messages by the receiving processor.

If more than two processors require "message" passing, multiple Mailboxes can be instantiated between the two processors. Each Intel FPGA Avalon Mailbox corresponds to one direction message passing.

13.2. Functional Description

Intel FPGA Avalon Mailbox provides two 32-bit registers for message passing between processors, Command register (0x0) and Pointer register (0x1). The message sender processor and message receiver processor have individual Avalon Memory Mapped (Avalon-MM) interfaces to a Mailbox component. A write to the command register by the sender processor indicates a pending message in the Mailbox and an interrupt will be issued to the receiver processor. Upon retrieval of the message by the receiver processor via a read transaction, the message is consumed, Mailbox is empty. The status register (0x2) is used to indicate if the Mailbox is full or empty.

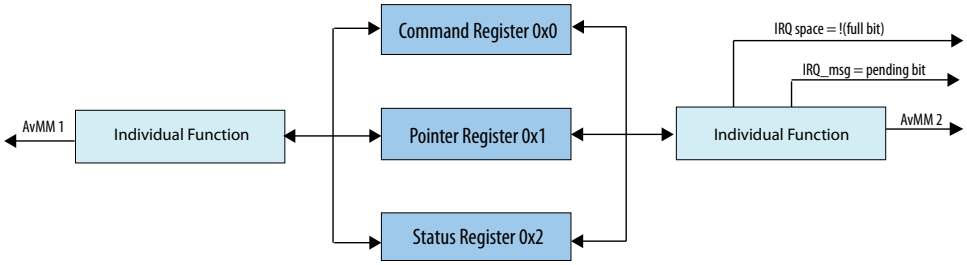
The Mailbox Avalon-MM interface which receives messages, or identified as sender interface, will back pressure the sender if there is message pending in the Mailbox. This will ensure every single message passed into the Mailbox is not overwritten. Upon message arrival, the receiving processor will then receive a level interrupt by the Mailbox. The interrupt will hold high until the single message is retrieved from the Mailbox via the Avalon-MM interface of receiving processor.

In addition, the Interrupt Masking Register (0x3) is writable by the Avalon-MM interface to mask its dedicated interrupt output. For example, receiver interface will be able to set the mask bit to mask off the message pending interrupt generated by Mailbox. Meanwhile, sender interface will be able to set the mask bit to mask off the message space interrupt output.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Figure 42. Intel FPGA Avalon Mailbox (simple) Block Diagram



The Mailbox is clocked with single source. Both of the Avalon-MM Agent interfaces have its individual function to set and clear the Full bit and Message Pending bit. The Avalon-MM Agent of the sender processor will only set the status bits, while the Avalon-MM Agent of the receiver processor only clears the status bit.

An interrupt is derived from the Status register bits. It will remain high until the message in the Mailbox is read.

13.2.1. Message Sending and Retrieval Process

The following are steps needed to send and receive messages through the Intel FPGA Avalon Mailbox (simple) component:

1. A process or host that intends to send a message will write to the Mailbox's Pointer register at address offset 0x1, then only to the Command register at address offset 0x0. Writing to the Command register indicates the completion of a message passing into the Mailbox.
2. When there is a message pending in the Mailbox, a level interrupt signal is issued to the processor that should receive the message. Optionally, the receiver processor may choose to poll the Status register at address offset 0x2 to determine if any message has arrived, if the interrupt signal is not used.
3. The process or host that needs to receive the message reads the Mailbox's Pointer register and then the Command register through the connected Avalon-MM interface. Upon reading of Command register, the message is considered delivered, and the Mailbox is empty.

13.2.2. Component Register Map

The following table illustrates the Mailbox registers map that is observed by each processor from its Avalon-MM interfaces.

Table 105. Mailbox Register Map

Word Address Offset	Register/ Queue Name	Width	Attribute	
			Sender	Receiver
0x0	Command register	32-bit	R/W	RO
0x1	Pointer register	32-bit	R/W	RO
0x2	Status register	32-bit	RO	RO
0x3	Interrupt Masking register	32-bit	R	R

continued...

Word Address Offset	Register/ Queue Name	Width	Attribute	
			Sender	Receiver
			Sender can only write to Message Space Interrupt Mask bit.	Receiver can only write to Message Pending Interrupt bit.

13.2.2.1. Command Register

The Command register is a 32-bit register which contains a user defined command to be passed between processors. This register is read-writeable via Avalon-MM Agent (sender). However it is only readable by the Avalon-MM Agent (receiver) interface.

13.2.2.2. Pointer Register

Instead of passing huge data via the Mailbox, a Pointer register is introduced. The Pointer register contains the 32-bit address to the payload of the message. A payload could be the raw data to be passed to the receiving processor for further processing. However, a message could contain zero payload or data for processing. A write to the Pointer may not be necessary for a message passing.

This register is read-writeable via Avalon-MM Agent (sender). However it is only readable by Avalon-MM Agent (receiver) interface.

13.2.2.3. Status Register

The Status register presents the full or empty status of the Mailbox. As the Mailbox can only contain one message at a time, the full bit status also indicates if there is message pending in the Mailbox. This register is read only by both Avalon-MM Agent interfaces.

Table 106. status Register Field

Bit Fields													
31	...		...		...		...		...		2	1	0
Reserved												Mailbox full	Message pending

Table 107. Mailbox status Register Descriptions

Filed Name	Description	Reset Value
Message pending	Value '0' indicates the Mailbox has no message. Value '1' indicates the Mailbox has message pending for retrieval.	0
Mailbox full	Value '1' indicates the Mailbox is full. Value '0' indicates Mailbox has space for incoming message.	0
Reserved	-	0

13.2.2.4. Interrupt Masking Register

The Interrupt Masking Register provides a masking bit to the Message Pending Interrupt and Message Space Interrupt. This register is accessible by both the sender and receiver of the Avalon-MM Agent interface. However, the editable bit is only

applicable for its corresponded interrupt. This means the sender Avalon-MM Agent can only modify the masking bit of Message Space Interrupt, whereas receiver Avalon-MM Agent can only modify the masking bit of Message Pending Interrupt. Read access of the whole register is available to both Avalon-MM Agent Interfaces.

Table 108. Interrupt Masking Register Field

Bit Fields													
31	...		...		...		...		...		2	1	0
Reserved												Message space mask	Message pending mask

Table 109. Interrupt Masking Register Descriptions

Filed Name	Description	Reset Value
Message pending mask	Value '0' to mask off the Message Pending Interrupt output. Value '1' enable Message Pending Interrupt upon triggered.	0
Mailbox space mask	Value '0' to mask off the Message Space Interrupt output. Value '1' enable Message Space Interrupt upon triggered.	0
Reserved	-	0

13.3. Interface

13.3.1. Component Interface

Intel FPGA Avalon Mailbox (simple) component consists of two Avalon-MM Agent interfaces, one dedicated for each processor. The Mailbox also provides active high level interrupt output, which is served as message arrival notification to the receiving processor. Optionally, a secondary IRQ is created as notification to the message sender indicating if Mailbox is available for incoming message.

Intel FPGA Avalon Mailbox (simple) has only one clock domain with one associated reset interface. Requirement of different clock domains between two processors is handled through the Platform Designer fabric. The following table describes the interfaces behavior of the component.

Table 110. Component Interface Behavior

Interface Port	Description	Details
Avalon MM Agent (sender)	Avalon-MM Agent interface for processor of message sender.	This interface apply wait request signal for back pressuring the Avalon-MM Host if Mailbox is already full.
Avalon MM Agent (receiver)	Avalon-MM Agent interface for processor of message receiver.	This interface only has read capability with readWaitTime=1.
Clock	Clock input of component.	It supports maximum frequency up to 400MHz on Cyclone IV and 600MHz in StratixIV devices.

continued...

Interface Port	Description	Details
Reset_n	Active LOW reset input/s.	Support asynchronous reset assertion. De-assertion of reset has to be synchronized to the input clock.
IRQ_msg	Message Pending Interrupt output to processor of message receiver upon message arrival. The signal will remain high until the message is retrieved.	Interrupt assertion and deassertion is synchronized to input clock.
IRQ_space	Message Space Interrupt output processor of message sender whenever Mailbox has space for incoming message. The signal will assert high as long as the Mailbox is yet full.	Interrupt assertion and deassertion is synchronized to input clock. The connection of this interrupt port to the top level is depends on configuration parameter of MSG_SPACE_NOTIFY.

13.3.2. Component Parameterization

Table 111. Intel FPGA Avalon Mailbox (simple) TCL Component Configuration Parameters

Parameter Name	Description	Default Value	Allowable Range
MSG_SPACE_NOTIFY	Boolean 'true' will enable interrupt output to message sending processor for indicating available space for incoming message	0	0, 1
MSG_ARRIVAL_NOTIFY	Boolean 'true' will enable interrupt output to message receiver processor for indicating a message is pending for retrieval.	1	0, 1

13.4. HAL Driver

This section describes the HAL driver for Intel FPGA Avalon Mailbox (simple) soft IP core. Intel FPGA Avalon Mailbox (simple) component provides a medium of communication between processors. It provides a message passing path between the sending processor and receiving processor. The receiver processor is notified through an interrupt upon message arrival or the driver will poll the status register if in polling mode. Intel FPGA Avalon Mailbox (simple) provides three 32-bit registers for message passing between processors, Command (0x0), Pointer (0x4), and Status register (0x8).

The driver code is located at:

```
/acds/main/ip/altera/altera_avalon_mailbox/hal/src/
altera_avalon_mailbox_simple.c
```

```
/acds/main/ip/altera/altera_avalon_mailbox/hal/inc/
altera_avalon_mailbox_simple.h
```

```
/acds/main/ip/altera/altera_avalon_mailbox/inc/
altera_avalon_mailbox_simple_regs.h
```

```
/acds/main/ip/altera/altera_avalon_mailbox/
altera_avalon_mailbox_simple_sw.tcl
```

13.4.1. Feature Description

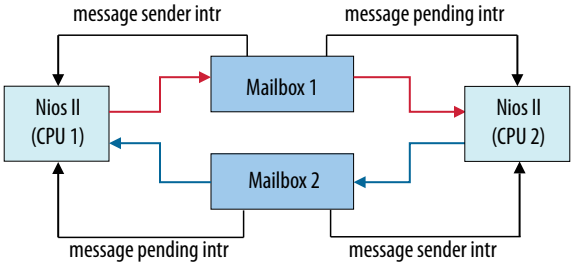
The Mailbox driver message delivery depends on how the Platform Designer design of the sender processor, receiver processor and Mailbox are interconnected. The Mailbox driver provides the features to send message to target processor and retrieve message for the receiver processor. The driver include an interrupt service routine when interrupt mode is used.

13.4.1.1. Component Configuration

13.4.1.1.1. Interrupt Mode

The figure below is an example of a design using the Intel FPGA Avalon Mailbox (simple) in interrupt mode. The sender CPU(1) will initiate a transfer of the message to the receiver CPU(2) by writing the command data to the Command register through Mailbox 1. The Command register will send a message pending interrupt to the receiver. The message pending interrupt is connected to the receiver CPU(2)'s IRQ to notify that a message has arrived. Once the Command register in Mailbox 1 is read, the message pending interrupt is cleared and the message is processed. On the sender CPU(1) side, once the message is read, a message sender interrupt will be flagged signaling that Mailbox 1 is free to transmit another message.

Figure 43. Example of a Bi-Directional Intel FPGA Avalon Mailbox System Using Interrupt Mode



13.4.1.1.2. Polling Mode

In the case of polling mode, you will always check on the Mailbox Status register if a message has arrived or free to send. Driver API functions include a timeout parameter, which allows you to specify whether a read or send operation must be completed within a certain period of time.

13.4.1.2. Driver Implementation

An opened Mailbox instance will register a sender/receiver interrupt service routine (ISR), if interrupts are supported with sender/receiver callbacks. When a Mailbox interrupt is disabled, an ISR will not register and polling mode will need to be used. You must close the Mailbox driver when it is unused.

Table 112. Mailbox APIs

Function Name	Description
altera_avalon_mailbox_send	Send message to Mailbox
altera_avalon_mailbox_status	Query current state of Mailbox
altera_avalon_mailbox_retrieve_poll	Read from Mailbox pointer register to retrieve messages
altera_avalon_mailbox_open	Claims a handle to a Mailbox, enabling all the other functions to access the Mailbox core
altera_avalon_mailbox_close	Close the handle to a Mailbox

Table 113. altera_avalon_mailbox_open

Prototype:	altera_avalon_mailbox_dev* altera_avalon_mailbox_open (const char* name, altera_mailbox_tx_cb tx_callback, altera_mailbox_rx_cb rx_callback)
Include:	<altera_avalon_mailbox_simple.h>
Parameters:	Name — The Mailbox device name to open. tx_callback – User to provide callback function to notify when a sending message is completed. rx_callback – User to provide callback function to notify when a receive a message.
Returns:	Pointer to mailbox
Description:	altera_avalon_mailbox_open() find and register the Mailbox device pointer. This function also registers the interrupt handler and user callback function for a interrupt enabled Mailbox.

Table 114. altera_avalon_mailbox_close

Prototype:	void altera_avalon_mailbox_close (altera_avalon_mailbox_dev* dev);
Include:	<altera_avalon_mailbox_simple.h>
Parameters:	dev—The Mailbox to close.
Returns:	Null
Description:	alt_avalon_mailbox_close() closes the mailbox de-registering interrupt handler and callback functions and masking Mailbox interrupt.

Table 115. altera_avalon_mailbox_send

Prototype:	int altera_avalon_mailbox_send (altera_avalon_mailbox_dev* dev, void* message, int timeout, EventType event)
Include:	<altera_avalon_mailbox_simple.h>
Parameters:	*message – Pointer to message command and pointer structure. Timeout – Specifies number of loops before sending a message. Give a '0' value to wait until the message is transferred. EventType – set 'POLL' or 'ISR'.
Returns:	Return 0 on success and 1 for fail.
Description:	altera_avalon_mailbox_send () sends a message to the mailbox. This is a blocking function when the sender interrupt is disabled. This function is in non-blocking when interrupt is enabled.

Table 116. altera_avalon_mailbox_retrieve_poll

Prototype:	int altera_avalon_mailbox_retrieve_poll (altera_avalon_mailbox_dev* dev,alt_u32* msg_ptr, alt_u32 timeout)
Include:	<altera_avalon_mailbox_simple.h>
Parameters:	dev - The Mailbox device to read message from. timeout – Specifies number loops before sending a message. Give a '0' value to wait until a message is retrieved. msg_ptr – A pointer to an array of two Dwords which are for the command and message pointer. This pointer will be populated with a receive message if successful or NULL if error.
Returns:	Return pointer to message and command. Return 'NULL' in messages if timeout. This is a blocking function.
Description:	altera_avalon_mailbox_retrieve_poll () reads a message pointer and command to Mailbox structure from the Mailbox and notifies through callback.

Table 117. altera_avalon_mailbox_status

Prototype:	alt_u32 altera_avalon_mailbox_status (altera_avalon_mailbox_dev* dev)
Include:	<altera_avalon_mailbox_simple.h>
Parameters:	dev -The Mailbox device to read status from
Returns:	For a receiving Mailbox: - 0 for no message pending - 1 for message pending For a sending Mailbox: - 0 for Mailbox empty (ready to send) - 1 for Mailbox full (not ready to send)
Description:	Indicates to sender Mailbox it is full or empty for transfer. Indicates to receiver Mailbox has a message pending or not.

Example 6. Device structure

```
// Callback routine type definition
typedef void(*altera_mailbox_rx_cb)(void *message);
typedefvoid (*altera_mailbox_tx_cb)(void *message,int status);

typedef enum mbox_type { MBOX_TX = 0,MBOX_RX } MboxType;
typedef enum event_type { ISR = 0, POLL } EventType;

typedef struct altera_avalon_mailbox_dev
{
    alt_dev                 dev;
    /* Device linke-list entry */

    alt_u32                 base;
    /* Base address of Mailbox */

    alt_u32                 mailbox_irq;
    /* Mailbox IRQ */

    alt_u32                 mailbox_intr_ctrl_id;
    /* Mailbox IRQ ID */

    altera_mailbox_tx_cb    tx_cb;
    /* Callback routine pointer */

    altera_mailbox_rx_cb    rx_cb;
    /* Callback routine pointer */

    MboxType                mbox_type;
    /* Mailbox direction */

    alt_u32*                mbox_msg;
    /* a pointer to message array to be received or sent */

    alt_u8                  lock;
    /* Token to indicate mbox_msg already taken */

    ALT_SEM                 (write_lock)
    /* Semaphore used to control access to the write in multi-threaded mode */
}
altera_avalon_mailbox_dev;
```

13.4.1.3. Driver Examples

The figure below demonstrates writing to a Mailbox. For this example, assume that the hardware system has two processors communicating via Mailboxes. The system includes two Mailbox cores, which provides two-way communication between the processors.

Example 7. Sender Processor Using Mailbox to Send a Message.

```
#include <stdio.h>
#include "altera_avalon_mailbox_simple.h"
#include "altera_avalon_mailbox_simple_regs.h"
#include "system.h"

/* example callback function from users*/
void tx_cb (void* report, int status) {
    if (!status) {
       printf (“Transfer done”);
    } else {
       printf (“error in transfer”);
    }

int main_sender()
```

```
{
alt_u32 message[2] = {0x00001111, 0xaa55aa55};
int timeout     = 50000;
alt_u32 status;
alt_avalon_mailbox_simple_dev* mailbox_sender;

/* Open mailbox on sender processor */
mailbox_sender = alt_avalon_mailbox_open("/dev/mailbox_simple_0", tx_cb,
NULL);

    if (!mailbox_sender){
        printf ("FAIL: Unable to open mailbox_simple");
        return 1;
        }

    /* Send a message to the other processor using interrupt */
    altera_avalon_mailbox_send (mailbox_sender, message, 0, ISR);

    /* Using polling method to send a message, with infinite timeout */
        timeout = 0;
        status = altera_avalon_mailbox_send (mailbox_sender, message, timeout,
POLL);

        if (status) {
            printf (“error in transfer”);
        } else {
            printf (“Transfer done”);
        }

    /* Closing mailbox device and de-registering interrupt handler and callback
*/
    altera_avalon_mailbox_close (mailbox_sender);
    return 0;
    }
```

Example 8. Receiver Processor Waiting for Message.

```
#include <stdio.h>
#include "altera_avalon_mailbox_simple.h"
#include "altera_avalon_mailbox_simple_regs.h"
#include "system.h"

void rx_cb (void* message) {
    /* Get message read from mailbox */
    alt_u32* data = alt_u32* message;
    if (message!= NULL) {
        printf (“Message received”);
    } else {
      printf (“Incomplete receive”);
    }

int main_receiver()
{
alt_u32* message[2];
int timeout     = 50000;
alt_avalon_mailbox_simple_dev* mailbox_rcv;

    /* This example is running on receiver processor */
    mailbox_rcv = alt_avalon_mailbox_open("/dev/mailbox_simple_1", NULL,
rx_cb);
    if (!mailbox_rcv){
        printf ("FAIL: Unable to open mailbox_simple");
        return 1;
    }

    /* For interrupt disable system */
    altera_avalon_mailbox_retrieve_poll (mailbox_rcv,message, timeout)

    if (message == NULL)
```

```
    printf (“Receive Error”);
    else
    printf (“Message received with Command 0x%x and Message 0x%x\n”,
message[0], message[1]);


altera_avalon_mailbox_close (mailbox_rcv);
return 0;
}
```

13.5. Intel FPGA Avalon Mailbox Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0	• Added register width in *Table: Mailbox Register Map*. • Corrected code location in *HAL Driver*.

Date	Version	Changes
November 2015	2015.11.06	Added HAL Driver section.
June 2015	2015.06.12	Initial release.

14. Intel FPGA Avalon Mutex Core

14.1. Core Overview

Multiprocessor environments can use the mutex core with Avalon interface to coordinate accesses to a shared resource. The mutex core provides a protocol to ensure mutually exclusive ownership of a shared resource.

The mutex core provides a hardware-based atomic test-and-set operation, allowing software in a multiprocessor environment to determine which processor owns the mutex. The mutex core can be used in conjunction with shared memory to implement additional interprocessor coordination features, such as mailboxes and software mutexes.

The mutex core is designed for use in Avalon-based processor systems, such as a Nios II processor system. Intel provides device drivers for the Nios II processor to enable use of the hardware mutex.

14.2. Functional Description

The mutex core has a simple Avalon Memory-Mapped (Avalon-MM) agent interface that provides access to two memory-mapped, 32-bit registers.

Table 118. Mutex Core Register Map

Offset	Register Name	R/W	Bit Description		
			31 16	15 1	0
0	mutex	RW	OWNER	VALUE	
1	reset	RW	Reserved		RESET

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

The mutex core has the following basic behavior. This description assumes there are multiple processors accessing a single mutex core, and each processor has a unique identifier (ID).

- When the VALUE field is `0x0000`, the mutex is unlocked and available. Otherwise, the mutex is locked and unavailable.
- The `mutex` register is always readable. Avalon-MM host peripherals, such as a processor, can read the `mutex` register to determine its current state.
- The `mutex` register is writable only under specific conditions. A write operation changes the `mutex` register only if one or both of the following conditions are true:
  - The VALUE field of the `mutex` register is zero.
  - The OWNER field of the `mutex` register matches the OWNER field in the data to be written.
- A processor attempts to acquire the mutex by writing its ID to the OWNER field, and writing a non-zero value to the VALUE field. The processor then checks if the acquisition succeeded by verifying the OWNER field.
- After system reset, the RESET bit in the `reset` register is high. Writing a one to this bit clears it.

14.3. Configuration

The MegaWizard™ Interface provides the following options:

- **Initial Value**—the initial contents of the VALUE field after reset. If the **Initial Value** setting is non-zero, you must also specify **Initial Owner**.
- **Initial Owner**—the initial contents of the OWNER field after reset. When **Initial Owner** is specified, this owner must release the mutex before it can be acquired by another owner.

14.4. Software Programming Model

The following sections describe the software programming model for the mutex core. For Nios II processor users, Intel provides routines to access the mutex core hardware. These functions are specific to the mutex core and directly manipulate low-level hardware. The mutex core cannot be accessed via the HAL API or the ANSI C standard library. In Nios II processor systems, a processor locks the mutex by writing the value of its `cpuid` control register to the OWNER field of the `mutex` register.

14.4.1. Software Files

Intel provides the following software files accompanying the mutex core:

- `altera_avalon_mutex_regs.h`—Defines the core's register map, providing symbolic constants to access the low-level hardware.
- `altera_avalon_mutex.h`—Defines data structures and functions to access the mutex core hardware.
- `altera_avalon_mutex.c`—Contains the implementations of the functions to access the mutex core

14.4.2. Hardware Access Routines

This section describes the low-level software constructs for manipulating the mutex core. The file `altera_avalon_mutex.h` declares a structure `alt_mutex_dev` that represents an instance of a mutex device. It also declares routines for accessing the mutex hardware structure, listed in the table below.

Table 119. Hardware Access Routines

Function Name	Description
`altera_avalon_mutex_open()`	Claims a handle to a mutex, enabling all the other functions to access the mutex core.
`altera_avalon_mutex_trylock()`	Tries to lock the mutex. Returns immediately if it fails to lock the mutex.
`altera_avalon_mutex_lock()`	Locks the mutex. Will not return until it has successfully claimed the mutex.
`altera_avalon_mutex_unlock()`	Unlocks the mutex.
`altera_avalon_mutex_is_mine()`	Determines if this CPU owns the mutex.
`altera_avalon_mutex_first_lock()`	Tests whether the mutex has been released since reset.

These routines coordinate access to the software mutex structure using a hardware mutex core. For a complete description of each function, see section the **Mutex API** section.

The code shown in below demonstrates opening a mutex device handle and locking a mutex.

```
#include <altera_avalon_mutex.h>

/* get the mutex device handle */
alt_mutex_dev* mutex = altera_avalon_mutex_open( “/dev/mutex” );

/* acquire the mutex, setting the value to one */
altera_avalon_mutex_lock( mutex, 1 );

/*
 * Access a shared resource here.
 */

/* release the lock */

altera_avalon_mutex_unlock( mutex );
```

14.5. Mutex API

This section describes the application programming interface (API) for the mutex core.

14.5.1. altera_avalon_mutex_is_mine()

Prototype:	`int altera_avalon_mutex_is_mine(alt_mutex_dev* dev)`
Thread-safe:	Yes.

continued...

Available from ISR:	No.
Include:	`<altera_avalon_mutex.h>`
Parameters:	`dev`—the mutex device to test.
Returns:	Returns non zero if the mutex is owned by this CPU.
Description:	`altera_avalon_mutex_is_mine()` determines if this CPU owns the mutex.

14.5.2. altera_avalon_mutex_first_lock()

Prototype:	`int altera_avalon_mutex_first_lock(alt_mutex_dev* dev)`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	`<altera_avalon_mutex.h>`
Parameters:	`dev`—the mutex device to test.
Returns:	Returns 1 if this mutex has not been released since reset, otherwise returns 0.
Description:	`altera_avalon_mutex_first_lock()` determines whether this mutex has been released since reset.

14.5.3. altera_avalon_mutex_lock()

Prototype:	`void altera_avalon_mutex_lock(alt_mutex_dev* dev, alt_u32 value)`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mutex.h**>
Parameters:	`dev`—the mutex device to acquire. `value`—the new value to write to the mutex.
Returns:	—
Description:	`altera_avalon_mutex_lock()` is a blocking routine that acquires a hardware mutex, and at the same time, loads the mutex with the `value` parameter.

14.5.4. altera_avalon_mutex_open()

Prototype:	`alt_mutex_dev* alt_hardware_mutex_open(const char* name)`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	`<altera_avalon_mutex.h>`
Parameters:	`name`—the name of the mutex device to open.
Returns:	A pointer to the mutex device structure associated with the supplied name, or NULL if no corresponding mutex device structure was found.
Description:	`altera_avalon_mutex_open()` retrieves a pointer to a hardware mutex device structure.

14.5.5. altera_avalon_mutex_trylock()

Prototype:	`int altera_avalon_mutex_trylock(alt_mutex_dev* dev, alt_u32 value)`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	`<altera_avalon_mutex.h>`
Parameters:	`dev`—the mutex device to lock. `value`—the new value to write to the mutex.
Returns:	`0` = The mutex was successfully locked. Others = The mutex was not locked.
Description:	`altera_avalon_mutex_trylock()` tries once to lock the hardware mutex, and returns immediately.

14.5.6. altera_avalon_mutex_unlock()

Prototype:	`void altera_avalon_mutex_unlock(alt_mutex_dev* dev)`
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_mutex.h**>
Parameters:	`dev`—the mutex device to unlock.
Returns:	Null.
Description:	`altera_avalon_mutex_unlock()` releases a hardware mutex device. Upon release, the value stored in the mutex is set to zero. If the caller does not hold the mutex, the behavior of this function is undefined.

14.6. Intel FPGA Avalon Mutex Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
July 2014	2014.07.24	Removed mention of SOPC Builder, updated to Platform Designer
December 2010	v10.1.0	Removed the “Device Support”, “Instantiating the Core in SOPC Builder”, and “Referenced Documents” sections.
July 2010	v10.0.0	No change from previous release.
November 2009	v9.1.0	No change from previous release.
March 2009	v9.0.0	No change from previous release.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	v8.0.0	No change from previous release.

15. Intel FPGA Avalon I²C (Host) Core

15.1. Core Overview

The Intel FPGA Avalon I²C (Host) core (*altera_avalon_i2c*) is an IP which implements the I²C protocol. It supports only host mode with a bit-rate up to 400 kbits/s (fast mode) and it can also operate in a multi-host system. It has an Avalon Memory-Mapped (Avalon-MM) agent interface for a host processor to access its control, status, command and data FIFO. Command and data FIFO can be configured to be accessible by either the Avalon-MM or Avalon Streaming (Avalon-ST). On the serial interface side, it provides two data and clock lines to communicate to remote I²C devices.

Note: This IP core supports Intel eASIC N5X devices.

15.2. Feature Description

15.2.1. Supported Features

- Supports I²C host mode
- Supports I²C standard mode (100 kbits/s) and fast mode (400 kbits/s)
- Supports multi-host operation
- Supports 7 bit or 10 bit addressing
- Supports START, repeated START and STOP generation
- Run time programmable SCL low and high period
- Interrupt or polled-mode of operation
- Avalon-MM agent interface for CSR registers access
- Avalon-MM or Avalon-ST for command and receive data FIFO access

15.2.2. Unsupported Features

I²C agent mode is not supported at the moment. Refer to Intel FPGA I²C Agent to Avalon MM Host Bridge IP for an I²C agent solution.

Related Information

Intel FPGA I2C Agent to Avalon-MM Host Bridge Core on page 196

15.3. Configuration Parameters

Configure the following parameters through Platform Designer.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Table 120. Platform Designer Parameters

Parameter	Legal Values	Default Values	Description
Interface for transfer command FIFO and receive data FIFO accesses	0 or 1	0	0: Avalon-MM interface access command and receive data FIFO 1: Avalon-ST interface access command and receive data FIFO
Depth of FIFO	4, 8, 16, 32, 64, 128, 256	4	Specify the Sizes of both the transfer command FIFO and the receive data FIFO

15.4. Interface

Figure 44. Intel FPGA Avalon I2C (Host) Core

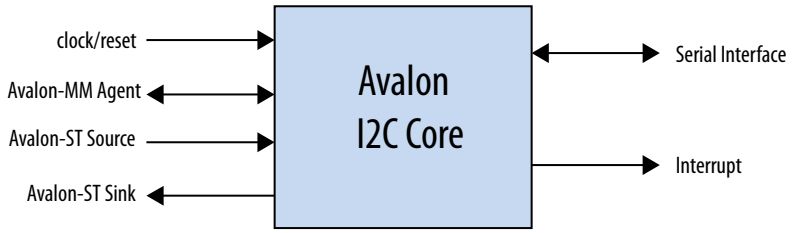


Table 121. Intel FPGA Avalon I2C (Host) Core Signals

Signal	Width	Direction	Description
Clock/Reset			
clk	1	Input	System clock source, Minimum clock frequency is 10 MHz.
rst_n	1	Input	System asynchronous reset source, *Note:* This signal is asynchronously asserted and synchronously de-asserted. The synchronous de-assertion must be provided externally to this peripheral.
Avalon-MM Agent			
addr	4	Input	Avalon-MM address bus. The address bus is in the unit of word addressing. For example, addr[2:0] = 0x0 is targeting the first word of the cores memory map space and addr[2:0] = 0x1 is targeting the second word.
read	1	Input	Avalon-MM read control
write	1	Input	Avalon-MM write control
readdata	32	Output	Avalon-MM read data bus
writedata	32	Input	Avalon-MM write data bus
Avalon-ST Source (20)			
src_data	8	Output	I2C data from receive data FIFO (RX_DATA)
src_valid	1	Output	Indicates src_data bus is valid
src_ready	1	Input	Indication from sink port that it is ready to consume src_data
Avalon-ST Sink(20)			

continued...

Signal	Width	Direction	Description
snk_data	10	Input	10-bit value driven by source port to transfer command FIFO (TFR_CMD)
snk_valid	1	Input	Indication from source port that snk_data is valid
snk_ready	1	Output	Indication from sink port that it is ready to consume snk_data
Serial Interface			
scl_oe	1	Output	Output enable for open drain buffer that drives SCL pin 1: SCL line pulled low 0: Open drain buffer is tri-stated and SCL line is externally pulled high
sda_oe	1	Output	Output enable for open drain buffer that drives SDA pin 1: SDA line pulled low 0: Open drain buffer is tri-stated and SDA line is externally pulled high
scl_in	1	Input	Input path of I²C's open drain buffer
sda_in	1	Input	It is from input path of I²C's open drain buffer
Interrupt			
intr	1	Output	Active high level interrupt output to host processor

15.5. Registers

15.5.1. Register Memory Map

Note: Each address offset represent 1 word of memory address space.

Table 122. Register Memory Map

Name	Address offset	Description
TFR_CMD	0x0	Transfer command FIFO
RX_DATA	0x1	Receive data FIFO
CTRL	0x2	Control register
ISER	0x3	Interrupt status enable register
ISR	0x4	Interrupt status register
STATUS	0x5	Status register
TFR_CMD_FIFO_LVL	0x6	TFR_CMD FIFO level register
RX_DATA_FIFO_LVL	0x7	RX_DATA FIFO level register

continued...

(20) These signals are not used if "Interface for transfer command FIFO and receive data FIFO accesses" is set to Avalon-MM Agent. This setting can be configured through Platform Designer.

Name	Address offset	Description
SCL_LOW	0x8	SCL low count register
SCL_HIGH	0x9	SCL high count register
SDA_HOLD	0xA	SDA hold count register

15.5.2. Register Descriptions

15.5.2.1. Transfer Command FIFO (TFR_CMD)

Table 123. Transfer Command FIFO (TFR_CMD)

Bit	Fields	Access	Default Value	Description
31:10	Reserved	N/A	0x0	Reserved
9	STA	W	N/A	1: Requests a repeated START condition to be generated before current byte transfer
8	STO	W	N/A	1: Requests a STOP condition to be generated after current byte transfer
7:1	AD	W	N/A	When in address phase, these fields act as address bits When in data phase with the core is configured as a host transmitter, these fields represent I²C data bit [7:1] of the data byte to be transmitted by the core. When in data phase and the core acts as host receiver, this field is not used
0	RW_D	W	N/A	When transfer is in address phase, this field is used to specify the direction of I²C transfer 0: Specifies I²C write transfer request 1: Specifies I²C read transfer request When transfer is in data phase with core is configured as a host transmitter, this field represents I²C data bit 0 of the data byte to be transmitted by the core. When transfer is in data phase and the core acts as host receiver, this field is not used

15.5.2.2. Receive Data FIFO (RX_DATA)

Table 124. Receive Data FIFO (RX_DATA)

Bit	Fields	Access	Default Value	Description
31:8	Reserved	N/A	0x0	Reserved
7:0	RXDATA	R	N/A	Byte received from I²C transfer

15.5.2.3. Control Register (CTRL)

Bit	Fields	Access	Default Value	Description
31:6	Reserved	N/A	0x0	Reserved
5:4	RX_DATA_FIFO_THD	R/W	0x0	Threshold level of the receive data FIFO *Note:* If the actual level is equal or more than the threshold level, RX_READY interrupt status bit is asserted. 0x3: RX_DATA FIFO is full 0x2: RX_DATA FIFO is ½ full 0x1: RX_DATA FIFO is ¼ full 0x0: 1 valid entry
3:2	TFR_CMD_FIFO_THD	R/W	0x0	Threshold level of the transfer command FIFO *Note:* If the actual level is equal or less than the threshold level, TX_READY interrupt status bit is asserted. 0x3: TFR_CMD FIFO is not full (has at least one empty entry) 0x2: TFR_CMD FIFO is ½ full 0x1: TFR_CMD FIFO is ¼ full 0x0: TFR_CMD FIFO is empty
1	BUS_SPEED	R/W	0x0	Bus speed 1: Fast mode (up to 400 kbits/s) 0: Standard mode (up to 100 kbits/s)
0	EN	R/W	0x0	The core enable bit 1: Core is enabled 0: Core is disabled

15.5.2.4. Interrupt Status Enable Register (ISER)

Table 125. Interrupt Status Enable Register (ISER)

Bit	Fields	Access	Default Value	Description
31:5	Reserved	N/A	0x0	Reserved
4	RX_OVER_EN	R/W	0x0	1: Enable interrupt for RX_OVER condition
3	ARBLOST_DET_EN	R/W	0x0	1: Enable interrupt for ARBLOST_DET condition
2	NACK_DET_EN	R/W	0x0	1: Enable interrupt for NACK_DET condition
1	RX_READY_EN	R/W	0x0	1: Enable interrupt for RX_READY condition
0	TX_READY_EN	R/W	0x0	1: Enable interrupt for TX_READY condition

15.5.2.5. Interrupt Status Register (ISR)

Table 126. Interrupt Status Register (ISR)

Bit	Fields	Access	Default Value	Description
31:5	Reserved	N/A	0x0	Reserved
4	RX_OVER	R/W1C	0x0	Receive overrun 1: Indicates receive data FIFO has overrun condition, new data is lost. *Note:* Writing 1 to this field clears the content of the field to 0.
3	ARBLOST_DET	R/W1C	0x0	Arbitration lost detected 1: Indicates core has lost the bus arbitration *Note:* Writing 1 to this field clears the content of this field to 0.
2	NACK_DET	R/W1C	0x0	No acknowledgement detected 1: Indicates NACK is received by the core *Note:* Writing 1 to this field clears the content of this field to 0.
1	RX_READY	R	0x0	Receive ready 1: Indicates receive data FIFO contains data sent by the remote I²C device. This bit is asserted when RX_DATA FIFO level is equal or more than RX_DATA FIFO threshold. *Note:* This field is automatically cleared by the core's hardware once the receive data FIFO level is less than RX_DATA FIFO threshold.
0	TX_READY	R	0x0	Transmit ready 1: Indicates transfer command FIFO is ready for data transmission. This bit is asserted when transfer command FIFO level is equal or less than TFR_CMD FIFO threshold. *Note:* This field is automatically cleared by the core's hardware once transfer command FIFO level is more than TFR_CMD FIFO threshold.

15.5.2.6. Status Register (STATUS)

Table 127. Status Register (STATUS)

Bit	Fields	Access	Default Value	Description
31:1	Reserved	N/A	0x0	Reserved
0	CORE_STATUS	R	0x0	Status of the core's state machine 1: State machine is not idle 0: State machine is idle

15.5.2.7. TFR CMD FIFO Level (TFR CMD FIFO LVL)

Table 128. TFR CMD FIFO Level (TFR CMD FIFO LVL)

Bit	Fields	Access	Default Value	Description
31: log2(FIFO_DEPTH) + 1	Reserved	N/A	0x0	Reserved
log2(FIFO_DEPTH) :0	LVL	R	0x0	Current level of TFR_CMD FIFO

15.5.2.8. RX Data FIFO Level (RX Data FIFO LVL)

Table 129. RX Data FIFO Level (RX Data FIFO LVL)

Bit	Fields	Access	Default Value	Description
31:log2(FIFO_DEPTH) + 1	Reserved	N/A	0x0	Reserved
log2(FIFO_DEPTH): 0	LVL	R	0x0	Current level of RX_DATA FIFO

15.5.2.9. SCL Low Count (SCL LOW)

Table 130. SCL Low Count (SCL LOW)

Bit	Fields	Access	Default Value	Description
31:16	Reserved	N/A	0x0	Reserved
15:0	COUNT_PERIOD	R/W	0x1	Low period of SCL in terms of number of clock cycles

15.5.2.10. SCL High Count (SCL HIGH)

Table 131. SCL High Count (SCL HIGH)

Bit	Fields	Access	Default Value	Description
31:16	Reserved	N/A	0x0	Reserved
15:0	COUNT_PERIOD	R/W	0x1	High period of SCL in term of number of clock cycles

15.5.2.11. SDA Hold Count (SDA HOLD)

Table 132. SDA Hold Count (SDA HOLD)

Bit	Fields	Access	Default Value	Description
31:16	Reserved	N/A	0x0	Reserved
15:0	COUNT_PERIOD	R/W	0x1	Hold period of SDA in term of number of clock cycles

15.6. Reset and Clock Requirements

The core is a single clock domain design. The frequency of the single clock source must be maintained throughout the run time period. This requirement is needed because the implementation of SCL low, SCL high, and SDA hold period is based on the frequency of the single clock source. If the frequency of the clock changes in the middle of the run time, the initial configuration of SCL low, SCL high and SDA hold period will be unable to produce the correct timing. On the next run, the system reconfigures those options to ensure correct timing is produced.

The core has a single reset input which is used to reset the entire core. The single reset input is required to be asynchronously asserted and synchronously de-asserted. The de-assertion of the reset must be synchronous to the single input clock source of the core. The reset synchronizer should be implemented externally.

15.7. Functional Description

15.7.1. Overview

The core implements I²C host functionality. It can generate all mandatory I²C transfer protocol through the TFR_CMD register configuration. The core supports a joint data streaming use-cases where the DMA core can be used for bulk transfer.

Figure 45. Intel FPGA Avalon I²C Host Core without DMA

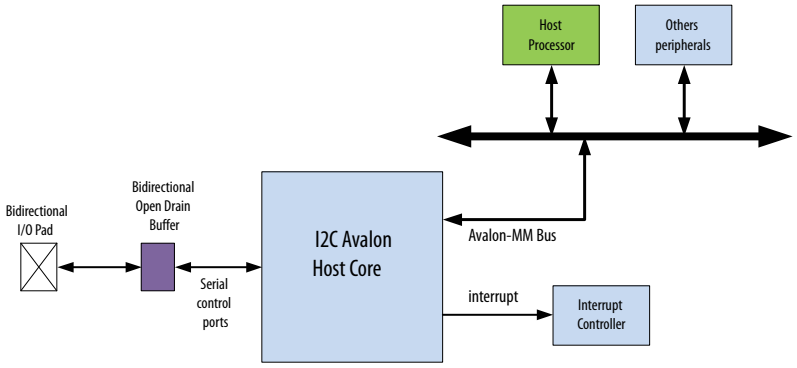
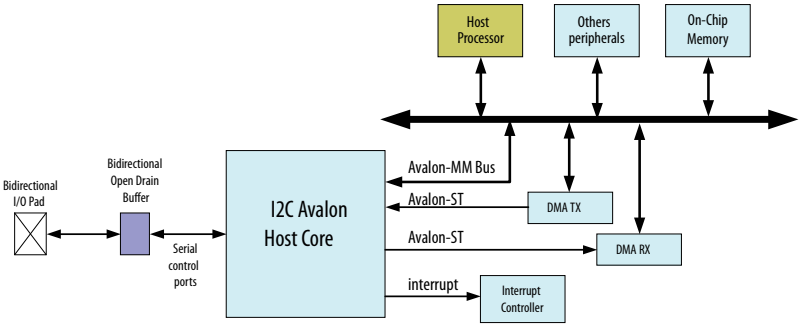


Figure 46. Intel FPGA Avalon I²C Host Core with DMA



15.7.2. Configuring TFT_CMD Register Examples

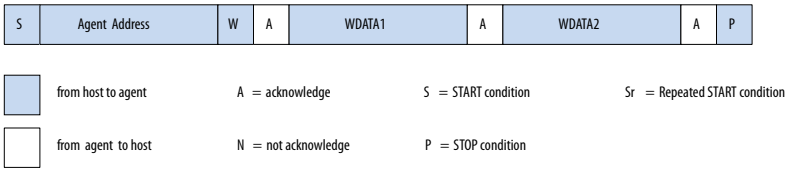
15.7.2.1. 7-bit Addressing Mode

Note: Assume the agent has a 7-bit address of 0x51.

15.7.2.1.1. Host Transmitter Writes 2 Bytes to Agent Receiver

Write `data1` = 0x55 and write `data2` = 0x56.

Figure 47. Host Transmitter Writes 2 Bytes to Agent Receiver



1. Writes TFR_CMD = 0x2A2 -> (STA = 0x1 , STOP = 0x0, AD = 0x51, RW_D = 0x0)
2. Writes TFR_CMD = 0x055 -> (STA = 0x0, STOP = 0x0, AD = 0x2A, RW_D = 0x1)
3. Writes TFR_CMD = 0x156 -> (STA = 0x0, STOP = 0x1, AD = 0x2B, RW_D = 0x0)

15.7.2.1.2. Host Receiver Reads 2 Bytes from Agent Transmitter

Read `data1` = 0x55 and read `data2` = 0x56.

Figure 48. Host Receiver Reads 2 Bytes from Agent Transmitter



1. Writes TFR_CMD = 0x2A3 -> (STA = 0x1, STOP = 0x0, AD = 0x51, RW_D = 0x1)
2. Writes TFR_CMD = 0x000 -> (STA = 0x0, STOP = 0x0, AD = 0x00, RW_D = 0x0)
3. Writes TFR_CMD = 0x100 -> (STA = 0x0, STOP = 0x1, AD = 0x00, RW_D = 0x0)

In steps 2 on page 184 and 3 on page 184, AD and RW_D fields are (don't care) and programmed to 0.

15.7.2.1.3. Combine Format (Host Writes 1 Byte and Changes Direction to Read 2 Bytes)

Write `data1` = 0x55, read `data1` = 0x56 and read `data2` = 0x57.

Figure 49. Combine Format (Host Writes 1 Byte and change direction to read 2 bytes)



1. Writes TFR_CMD = 0x2A2 -> (STA = 0x1 , STOP = 0x0, AD = 0x51, RW_D = 0x0)
2. Writes TFR_CMD = 0x055 -> (STA = 0x0, STOP = 0x0, AD = 0x2A, RW_D = 0x1)
3. Writes TFR_CMD = 0x2A3 -> (STA = 0x1, STOP = 0x0, AD = 0x51, RW_D = 0x1)
4. Writes TFR_CMD = 0x000 -> (STA = 0x0, STOP = 0x0, AD = 0x00, RW_D = 0x0)
5. Writes TFR_CMD = 0x100 -> (STA = 0x0, STOP = 0x1, AD = 0x00, RW_D = 0x0)

15.7.2.2. 10-bit Addressing Mode

Note: Assume agent 10-bit address = 0x351.

15.7.2.2.1. Host Transmitter Writes 2 Bytes to Agent Receiver

Write `data1` = 0x55 and write `data2` = 0x56.

Figure 50. Host Transmitter Writes 2 Bytes to Agent Receiver



1. Writes TFR_CMD = 0x2F6 -> (STA = 0x1, STOP = 0x0, AD = 0x7B, RW_D = 0x0)
2. Writes TFR_CMD = 0x051 -> (STA = 0x0, STOP = 0x0, AD = 0x28, RW_D = 0x1)
3. Writes TFR_CMD = 0x055 -> (STA = 0x0, STOP = 0x0, AD = 0x2A, RW_D = 0x1)
4. Writes TFR_CMD = 0x156 -> (STA = 0x0, STOP = 0x1, AD = 0x2B, RW_D = 0x0)

15.7.2.2.2. Host Receiver Reads 2 Bytes from Agent Transmitter

Read `data1` = 0x55 and read `data2` = 0x56.

Figure 51. Host Receiver Reads 2 Bytes from Agent Transmitter



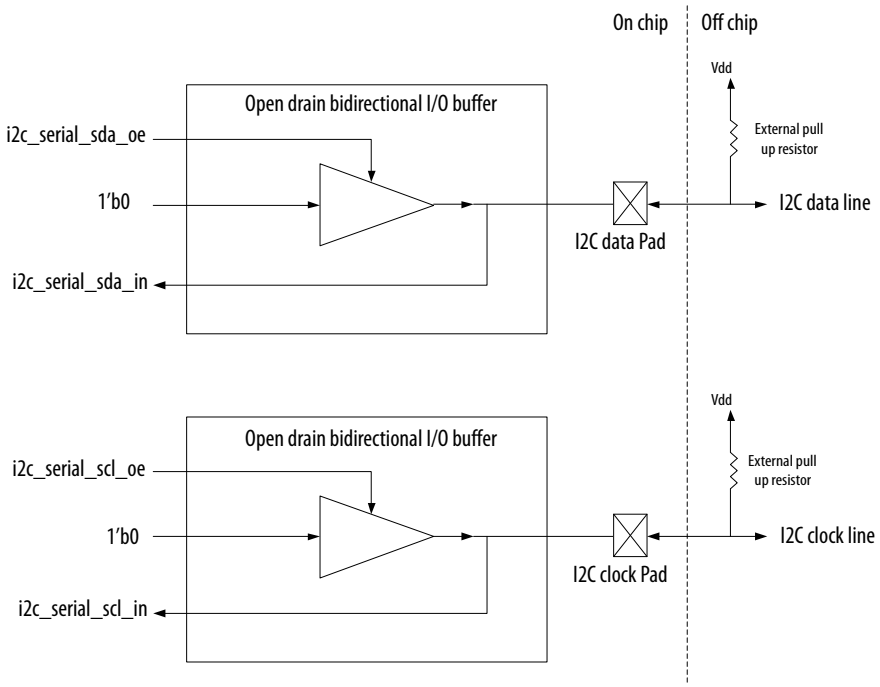
1. Writes TFR_CMD = 0x2F6 -> (STA = 0x1, STOP = 0x0, AD = 0x7B, RW_D = 0x0)
2. Writes TFR_CMD = 0x051 -> (STA = 0x0, STOP = 0x0, AD = 0x28, RW_D = 0x1)
3. Writes TFR_CMD = 0x2F7 -> (STA = 0x1, STOP = 0x0, AD = 0x7B, RW_D = 0x1)
4. Writes TFR_CMD = 0x000 -> (STA = 0x0, STOP = 0x0, AD = 0x00, RW_D = 0x0)
5. Writes TFR_CMD = 0x100 -> (STA = 0x0, STOP = 0x1, AD = 0x00, RW_D = 0x0)

In steps 4 on page 185 and 5 on page 185, AD and RW_D fields are (don't care) and programmed to 0.

15.7.3. I²C Serial Interface Connection

The core provides four ports for I²C serial connections. For external I²C serial connections, both `sda_in` and `sda_oe` are connected to a bidirectional open drain I²C data line buffer. Both `scl_in` and `scl_oe` are connected to another bidirectional open drain I²C clock line buffer. It is recommended to use the I/O IP core to generate the bidirectional open drain buffer. You can then instantiates the generated buffer primitives from the IP core into their system top level design file.

Figure 52. I²C Serial Interface Connection




```
Sample Code for I²C Serial Interface Connection

Verilog:

assign i2c_serial_scl_in = arduino_adc_scl;
assign arduino_adc_scl = i2c_serial_scl_oe ? 1'b0 : 1'bz;

assign i2c_serial_sda_in = arduino_adc_sda;
assign arduino_adc_sda = i2c_serial_sda_oe ? 1'b0 : 1'bz;

VHDL:
```

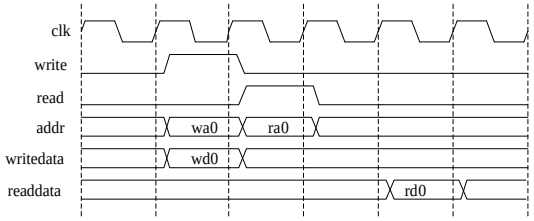
```
i2c_scl_in <= arduino_adc_scl;
arduino_adc_scl  <= '0' when i2c_scl_oe = '1' else 'Z';
i2c_sda_in <= arduino_adc_sda;
arduino_adc_sda  <= '0' when i2c_sda_oe = '1' else 'Z';
```

15.7.4. Avalon-MM Agent Interface

The Avalon-MM agent interface is configured as follows:

- Bus width: 32-bit
- Burst support: No
- Fixed read & write wait time: 0 cycles
- Fixed read latency: 2 cycles
- Lock support: No

Figure 53. Write and Read Timing of Avalon-MM Agent Interface



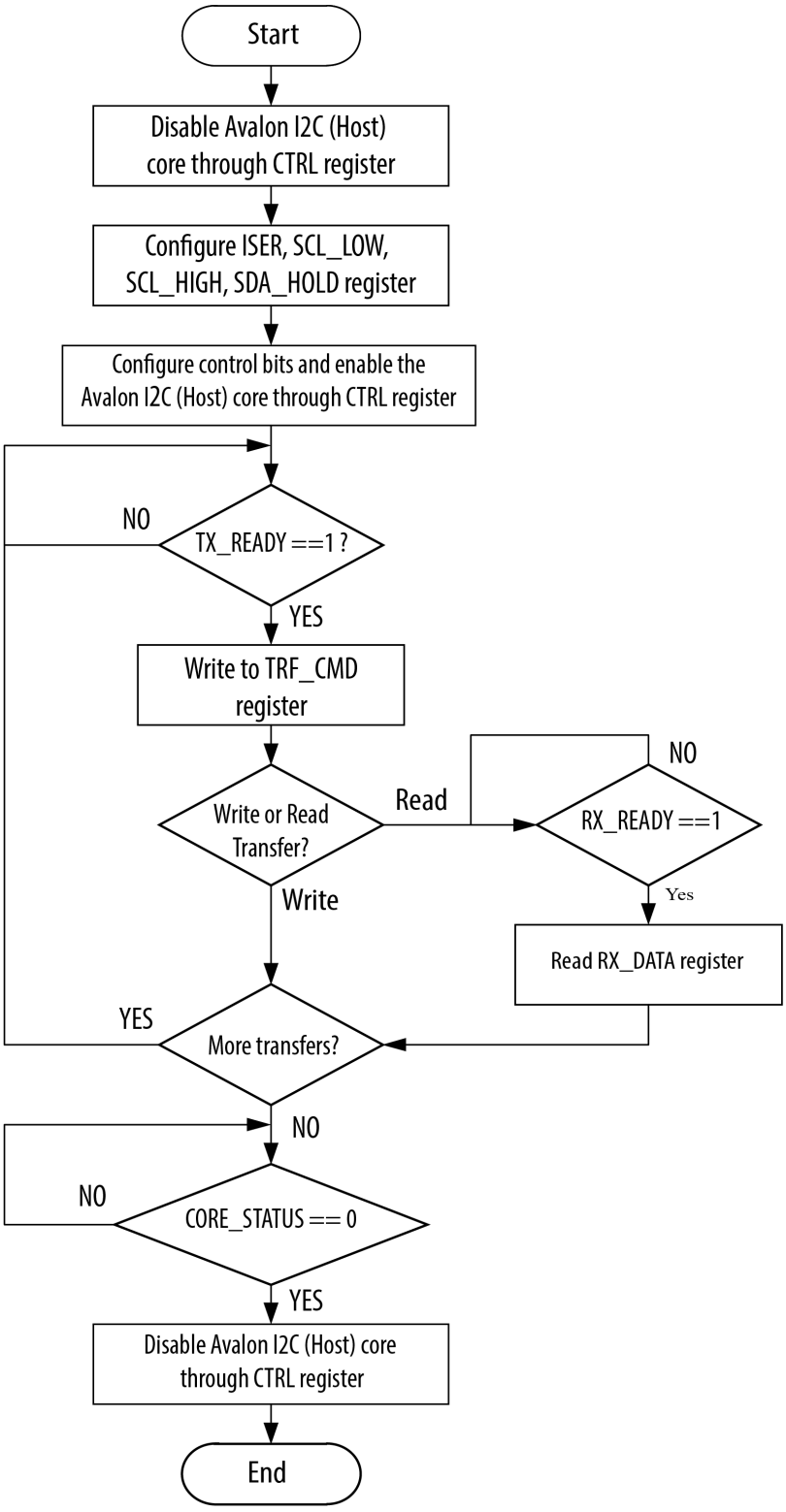
15.7.5. Avalon-ST Interface

Both ST data source and ST data sink interfaces support a ready latency of zero.

15.7.6. Programming Model

The following flowchart illustrates the recommended programming flow for the core.

Figure 54. Programming Model Flowchart



Note: When either ARBLOST_DET or NACK_DET occur, you need to clear its respective interrupt status register bits in their error handling procedure before continuing with a new I²C transfer. A new I²C transfer can be initiated with or without disabling the core.

Illustration: How to use the API

```
int main()
{
  ALT_AVALON_I2C_DEV_t *i2c_dev;  //pointer to instance structure
  alt_u8 txbuffer[0x200];
  alt_u8 rxbuffer[0x200];
  int i;
  ALT_AVALON_I2C_STATUS_CODE status;

  //get a pointer to the avalon i2c instance
  i2c_dev = alt_avalon_i2c_open("/dev/i2c_0");
  if (NULL==i2c_dev)
  {
      printf("Error: Cannot find /dev/i2c_0\n");
      return 1;
  }

  //set the address of the device using

  alt_avalon_i2c_master_target_set(i2c_dev,0x51)

  //write data to an eeprom at address 0x0200

  txbuffer[0]=2; txbuffer[1]=0;

  //The eeprom address which will be sent as first two bytes of data

  for (i=0;i<0x10;i++) txbuffer[i+2]=i;   //some data to write
  status=alt_avalon_i2c_master_tx(i2c_dev,txbuffer, 0x10+2,
ALT_AVALON_I2C_NO_INTERRUPTS);
  if (status!=ALT_AVALON_I2C_SUCCESS) return 1; //FAIL

  //read back the data into rxbuffer
  //This command sends the 2 byte eeprom data address required by the eeprom
  //Then does a restart and receives the data.
  status=alt_avalon_i2c_master_tx_rx(i2c_dev, txbuffer, 2, rxbuffer, 0x10,
ALT_AVALON_I2C_NO_INTERRUPTS);
  if (status!=ALT_AVALON_I2C_SUCCESS) return 1; //FAIL
  return 0;
}
```

```
//Using the optional irq callback:

int main()
{
  ALT_AVALON_I2C_DEV_t *i2c_dev;  //pointer to instance structure
  alt_u8 txbuffer[0x210];
  alt_u8 rxbuffer[0x200];
  int i;
  ALT_AVALON_I2C_STATUS_CODE status;

  //storage for the optional provided interrupt handler structure
  IRQ_DATA_t irq_data;

  //get a pointer to the avalon i2c instance
  i2c_dev = alt_avalon_i2c_open("/dev/i2c_0");
  if (NULL==i2c_dev)
  {
      printf("Error: Cannot find /dev/i2c_0\n");
      return 1;
  }
```

```
  //register the optional interrupt callback.
  alt_avalon_i2c_register_optional_irq_handler(i2c_dev,&irq_data);

  //set the address of the device we will be using
  alt_avalon_i2c_master_target_set(i2c_dev,0x51);

  //assume an eeprom at address 0x51

  //write data to an eeprom at address (within the eeprom) 0x0200
  txbuffer[0]=2;
  txbuffer[1]=0;

  //The eeprom data address which will be sent as first two bytes of data

  for (i=0;i<0x10;i++) txbuffer[i+2]=i;   //some data to write

  while (1) {  //for function retry
     status=alt_avalon_i2c_master_tx(i2c_dev, txbuffer, 0x10+2,
ALT_AVALON_I2C_USE_INTERRUPTS);


     if (status!=ALT_AVALON_I2C_SUCCESS) return 1; //FAIL

     //Completion should be checked by using the
alt_avalon_i2c_interrupt_transaction_status function.
     //Note: Interrupt and non-interrupt transactions can be mixed in any
sequence, so if desired this short address setup transaction can use
ALT_AVALON_I2C_NO_INTERRUPTS.


     while (alt_avalon_i2c_interrupt_transaction_status(i2c_dev) ==
ALT_AVALON_I2C_BUSY) {  }

     //Did the transaction complete OK? If yes then break out of this retry
loop, otherwise, have to do the transaction again
     //You may want to have a retry limit instead of

     while (1)
     if (alt_avalon_i2c_interrupt_transaction_status(i2c_dev) ==
ALT_AVALON_I2C_SUCCESS) break;
  }
  while (1) {

     //for function retry, read back the data into rxbuffer


     status=alt_avalon_i2c_master_tx_rx(i2c_dev, txbuffer, 2, rxbuffer, 0x10,
ALT_AVALON_I2C_USE_INTERRUPTS);


     if (status!=ALT_AVALON_I2C_SUCCESS) return 1; //FAIL

     //For this example we will just waste the time in a loop.


     while (alt_avalon_i2c_interrupt_transaction_status(i2c_dev) ==
ALT_AVALON_I2C_BUSY) {  }

     //Did the transaction complete OK

     if (alt_avalon_i2c_interrupt_transaction_status(i2c_dev) ==
ALT_AVALON_I2C_SUCCESS) break;
  }

  return 0;
}
```

15.8. Intel FPGA Avalon I2C (Host) Core API

Table 133. alt_avalon_i2c_open

Prototype	ALT_AVALON_I2C_DEV_t* alt_avalon_i2c_open(const char* name)
Include	<altera_avalon_i2c.h>
Parameters	• name - Character pointer to name of I2C peripheral as registered with the HAL. For example: To open an I2C controller named as `i2c_0` in Platform Designer, use `/dev/i2c_0`.
Returns	• NULL - The return value is NULL on failure. • non-NULL - Success, returns ptr to I2C device instance.
Description	Retrieve a pointer to the I2C block instance. Search the list of registered I2C instances for one with the supplied name.

Table 134. alt_avalon_i2c_register_optional_irq_handler

Prototype	void alt_avalon_i2c_register_optional_irq_handler(ALT_AVALON_I2C_DEV_t *i2c_dev, IRQ_DATA_t * irq_data)
Include	<altera_avalon_i2c.h>
Parameters	• i2c_dev – Pointer to I2C device (instance) structure. • irq data – A structure used for interrupt handler data.
Returns	-
Description	Associate the optionally provided user interrupt callback routine with the I2C handler. This is a simple IRQ callback which allows I2C transaction functions to immediately return while the optional callback handles receiving or transmitting the data to the device and completing the transaction. This optional callback uses a IRQ_DATA_t structure for IRQ data. You can use the function `alt_avalon_i2c_interrupt_transaction_status` to check for IRQ transaction complete, or for a transaction error. These optionally provided interrupt routines are functional, but are provided mainly for the purpose as working examples of using interrupts with the Avalon I2C (Host) Core. You may want to develop a more detailed irq callback routine tailored for specific device hardware. In that case, you shall register the user callback with the `alt_avalon_i2c_register_callback` function. Using this optionally provided user callback routine enables use of: • `alt_avalon_i2c_master_receive_using_interrupts` • `alt_avalon_i2c_master_transmit_using_interrupts functions`

Table 135. alt_avalon_i2c_register_callback

Prototype	void alt_avalon_i2c_register_callback (ALT_AVALON_I2C_DEV_t *i2c_dev, alt_avalon_i2c_callback callback, alt_u32 control, void *context)
Include	<altera_avalon_i2c.h>
Parameters	• dev - Pointer to I2C device (instance) structure. • callback - Pointer to callback routine to execute at interrupt level. • control - For masking the source interruption and setting configuration. • context - Callback context.
Returns	-
Description	Associate a user-specific routine with the I2C interrupt handler. If a callback is registered, all enabled ISR's causes the callback to be executed. The callback runs as part of the interrupt service routine. An optional user callback routine is provided in this code and, if used, enables use of:

continued...

	• alt_avalon_i2c_master_receive_using_interrupts • alt_avalon_i2c_master_transmit_using_interrupts functions To register the optionally provided user callback use the alt_avalon_i2c_register_optional_irq_handler function.

Table 136. alt_avalon_i2c_master_tx

Prototype	alt_avalon_i2c_master_tx (ALT_AVALON_I2C_DEV_t *i2c_dev, alt_u8 * buffer, const alt_u32 size, const alt_u8 use_interrupts)
Include	<altera_avalon_i2c.h>
Parameters	• dev - Pointer to I²C device (instance) structure. • buffer - The data buffer to transmit the I2C data. • size - The size of the data buffer to write to the I2C bus. • use_interrupts - The optional user interrupt handler callback is used to handle sending the data.
Returns	ALT_AVALON_I2C_SUCCESS indicates successful status. Otherwise, one of the ALT_AVALON_I2C_* status codes is returned. All failing return values are < 0.
Description	This function transmits START followed by the I2C command byte(s). Then write requests are sent to fulfill the write request. The final transaction will issue a STOP. This API is not suitable for being called in an interrupt context as it may wait for certain controller states before completing. The target address must have been set before using this function. If an ALT_AVALON_I2C_ARB_LOST_ERR, ALT_AVALON_I2C_NACK_ERR, or ALT_AVALON_I2C_BUSY occurs, then the command is retried. If the use_interrupts parameter is 1, then as soon as all cmd bytes have been written to the command fifo this function will return. The interrupt handler will then handle waiting for the device to send the rx data and will then complete the I2C transaction. To use this option the provided optional user interrupt handler callback must have been registered by calling the alt_avalon_i2c_register_optional_irq_handler function.

Table 137. alt_avalon_i2c_master_rx

Prototype	alt_avalon_i2c_master_rx (ALT_AVALON_I2C_DEV_t *i2c_dev, alt_u8 * buffer, const alt_u32 size, const alt_u8 use_interrupts)
Include	<altera_avalon_i2c.h>
Parameters	• dev - Pointer to I²C device (instance) structure. • buffer - The data buffer to receive the I2C data. • size - The size of the data buffer to write to the I2C bus. • use_interrupts - The optional user interrupt handler callback is used to handle receiving the data.
Returns	ALT_AVALON_I2C_SUCCESS indicates successful status. Otherwise, one of the ALT_AVALON_I2C_* status codes is returned. All failing return values are < 0.
Description	This function transmits START followed by the I2C command byte(s). Then read requests are sent to fulfill the read request. The final transaction will issue a STOP. This API is not suitable for being called in an interrupt context as it may wait for certain controller states before completing. The target address must have been set before using this function. If an ALT_AVALON_I2C_ARB_LOST_ERR, ALT_AVALON_I2C_NACK_ERR, ALT_AVALON_I2C_BUSY occurs, the command will be retried. ALT_AVALON_I2C_NACK_ERR will occur when the agent device is not yet ready to accept more data. If the use_interrupts parameter is 1, then as soon as all cmd bytes have been written to the command fifo, this function will return. The interrupt handler will then handle waiting for the device to send the rx data and will then complete the I2C transaction. To use this option the provided optional user interrupt handler callback must have been registered by calling the alt_avalon_i2c_register_optional_irq_handler function.

Table 138. alt_avalon_i2c_master_tx_rx

Prototype	alt_avalon_i2c_master_tx_rx (ALT_AVALON_I2C_DEV_t *i2c_dev, const alt_u8 * txbuffer, const alt_u32 txsize, alt_u8 * rxbuffer, const alt_u32 rxsize, const alt_u8 use_interrupts)
Include	<altera_avalon_i2c.h>
Parameters	• dev - Pointer to I²C device (instance) structure. • txdata - Send data buffer. • txsize - Size of the send data buffer to write to the I2C bus. • rxdata - Receive data buffer. • rxsize - Size of the receive data buffer. • use_interrupts - The optional user interrupt handler callback is used to handle sending and receiving the data.
Returns	ALT_AVALON_I2C_SUCCESS indicates successful status. Otherwise, one of the ALT_AVALON_I2C_* status codes is returned. All failing return values are < 0.

Description	This function transmits START followed by the I2C command byte(s). Then write requests are sent to fulfill the write request. After a RESTART is issued and read requests are sent until the read request is fulfilled. The final transaction will issue a STOP.
	ALT_AVALON_I2C_ARB_LOST_ERR, ALT_AVALON_I2C_NACK_ERR, or This API is not suitable for being called in an interrupt context as it may wait for certain controller states before completing. The target address must have been set before using this function. If an ALT_AVALON_I2C_BUSY occurs, then the command is retried.
	ALT_AVALON_I2C_NACK_ERR occurs when the agent device is not yet ready to accept or send more data. This command allows easy access of a device requiring an internal register address to be set before doing a read, for example an EEPROM device.
	Example: If an EEPROM requires a 2 byte address to be sent before doing a memory read, the tx buffer contains the 2 byte address and the txsize is set to 2. Then the rxbuffer recieves the rxsize number of bytes to read from the EEPROM as follows:
	`To Read 0x10 bytes from EEPROM at I2C address 0x51 into buffer:`
	`buffer[0]=2;buffer[1]=0; //set eeprom address 0x200`
	`alt_avalon_i2c_master_tx_rx(i2c_ptr,buffer,2,buffer,0x10,0);`
	The tx and rx buffer can be the same buffer if desired.
	If the use_interrupts parameter is 1, then as soon as all cmd bytes have been written to the command fifo this function will return. The interrupt handler will then handle waiting for the device to send the rx data and will then complete the I2C transaction. To use this option the provided optional user interrupt handler callback must have been registered by calling the alt_avalon_i2c_register_optional_irq_handler function.

Table 139. alt_avalon_i2c_interrupt_transaction_status

Prototype	This API is not suitable for being called in analt_avalon_i2c_interrupt_transaction_status (ALT_AVALON_I2C_DEV_t *i2c_dev)
Include	<altera_avalon_i2c.h>
Parameters	dev - A pointer to the I2C controller device block instance.
Returns	ALT_AVALON_I2C_SUCCESS indicates interrupt transaction is successfully completed. Another transaction can now be started. ALT_AVALON_I2C_BUSY indicates the interrupt transaction is still busy. ALT_AVALON_NACK_ERROR indicates the device did not ack. This is most likely because the device is busy with the previous transaction. The transaction must be retried. Otherwise, one of the other ALT_AVALON_I2C_* status codes is returned. The transaction must be retried. All failing return values are < 0.
Description	When an interrupt transaction has been initiated using the alt_avalon_i2c_master_tx, alt_avalon_i2c_master_rx, or alt_avalon_i2c_master_tx_rx function with the interrupt option set, or if using the alt_avalon_i2c_master_transmit_using_interrupts or alt_avalon_i2c_master_receive_using_interrupts functions, then this function can be used to check the status of that transaction. The only way to ensure error free back to

continued...

	back transactions is to use this function after every interrupt transaction to ensure the transaction had no errors and is complete, before starting the next transaction. Also, if an error is returned from this function, then you must retry the I2C transaction. One reason an error may be returned is if the device is busy, which is likely to occur occasionally if you are doing back to back transactions.

15.8.1. Optional Status Retrieval API

Table 140. alt_avalon_i2c_master_target_get

Prototype	void alt_avalon_i2c_master_target_get (ALT_AVALON_I2C_DEV_t * i2c_dev, alt_u32 * target_addr)
Include	<altera_avalon_i2c.h>
Parameters	• i2c_dev - A pointer to the I2C controller device block instance. • target_addr - The 7 or 10 bit agent target address.
Returns	-
Description	This function returns the current target address.

Table 141. alt_avalon_i2c_master_config_get

Prototype	void alt_avalon_i2c_master_config_get (ALT_AVALON_I2C_DEV_t *i2c_dev, ALT_AVALON_I2C_MASTER_CONFIG_t* cfg)
Include	<altera_avalon_i2c.h>
Parameters	• i2c_dev - A pointer to the I2C controller device block instance. • cfg - Pointer to a `ALT_AVALON_I2C_MASTER_CONFIG_t` structure for holding the returned I2C host mode configuration parameters.
Returns	-
Description	Populates the host mode configuration structure (type `ALT_AVALON_I2C_ADDR_MODE_t`) from registers.

Table 142. alt_avalon_i2c_master_config_speed_get

Prototype	alt_avalon_i2c_master_config_speed_get (ALT_AVALON_I2C_DEV_t *i2c_dev, const ALT_AVALON_I2C_MASTER_CONFIG_t* cfg, alt_u32 * speed_in_hz)
Include	<altera_avalon_i2c.h>
Parameters	• i2c_dev - A pointer to the I2C controller device block instance. • cfg - Pointer to a `ALT_AVALON_I2C_MASTER_CONFIG_t` structure for holding the returned I2C host mode configuration parameters. • speed_in_hz - Speed (Hz) the I2C bus is currently configured at based on the cfg structure (not necessarily on the hardware settings). To get the hardware speed first populate the cfg structure with the `alt_avalon_i2c_master_config_get()` function.
Returns	`ALT_AVALON_I2C_SUCCESS` - Indicates successful status. Otherwise, one of the `ALT_AVALON_I2C_*` status codes is returned. All failing return values are < 0.
Description	This utility function returns the speed in hertz (Hz) based on the contents of the passed in configuration structure.

Table 143. alt_avalon_i2c_int_status_get

Prototype	void alt_avalon_i2c_int_status_get (ALT_AVALON_I2C_DEV_t *i2c_dev, alt_u32 *status)
Include	<altera_avalon_i2c.h>
	continued...

Parameters	• i2c_dev - A pointer to the I2C controller device block instance. • status - A pointer to a bit mask of the active \ref ALT_AVALON_I2C_STATUS_t interrupt and status conditions.
Returns	-
Description	This function returns the current value of the I2C controller interrupt status register value which reflects the current I2C controller status conditions that are not disabled (or masked).

Table 144. alt_avalon_i2c_int_raw_status_get

Prototype	void alt_avalon_i2c_int_raw_status_get (ALT_AVALON_I2C_DEV_t *i2c_dev, alt_u32 *status)
Include	<altera_avalon_i2c.h>
Parameters	• i2c_dev - A pointer to the I2C controller device block instance. • status - A pointer to a bit mask of the active \ref ALT_AVALON_I2C_STATUS_t interrupt and status conditions.
Returns	-
Description	This function returns the current value of the I2C controller raw interrupt status register value which reflects the current I2C controller status conditions regardless of whether they are disabled/masked or not.

Table 145. alt_avalon_i2c_tfr_cmd_fifo_threshold_get

Prototype	void alt_avalon_i2c_tfr_cmd_fifo_threshold_get (ALT_AVALON_I2C_DEV_t *i2c_dev, ALT_AVALON_I2C_TFR_CMD_FIFO_THRESHOLD_t *threshold)
Include	<altera_avalon_i2c.h>
Parameters	• i2c_dev - A pointer to the I2C controller device block instance. • threshold - The current threshold value.
Returns	-
Description	Gets the current Transfer Command FIFO threshold level value.

Table 146. alt_avalon_i2c_rx_fifo_threshold_get

Prototype	
Include	<altera_avalon_i2c.h>void alt_avalon_i2c_rx_fifo_threshold_get (ALT_AVALON_I2C_DEV_t *i2c_dev, ALT_AVALON_I2C_RX_DATA_FIFO_THRESHOLD_t *threshold)
Parameters	• i2c_dev - A pointer to the I2C controller device block instance. • threshold - The current threshold value.
voidReturns	-
Description	Gets the current receive FIFO threshold level value.

15.9. Intel FPGA Avalon I²C (Host) Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2021.03.29	21.1	Added support for Intel eASIC N5X devices.
2020.12.23	20.3	Added minimum clock frequency for Intel FPGA Avalon I²C (Host) Core in section: *Interface*.
2018.09.24	18.1	Added a new section: *Intel FPGA Avalon I²C (Host) Core API*.
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
November 2017	2017.11.06	• Added the sample verilog code for *I²C Serial Interface Connection*. • Added the illustration for using API in *Programming Model*.
October 2016	2016.10.28	Initial release

16. Intel FPGA I²C Agent to Avalon-MM Host Bridge Core

16.1. Core Overview

The Intel FPGA I²C Agent to Avalon-MM Host Bridge soft IP core is a solution to connect an I²C interface with a User Flash Memory (UFM) device. This IP translates an I²C transaction into an Avalon Memory Mapped (Avalon-MM) transaction.

Note: This IP core supports Intel eASIC N5X devices.

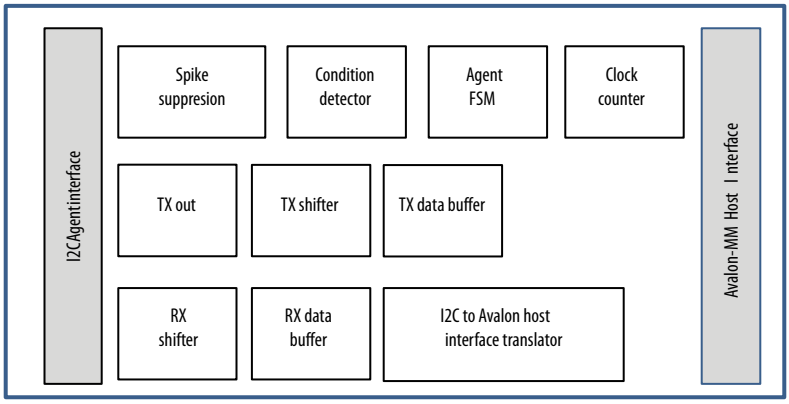
16.2. Functional Description

The I²C Agent to Avalon-MM Host Bridge core has the following features:

- Up to 4-byte addressing mode
- 3-bit address stealing
- 7-bit address I²C agent

16.2.1. Block Diagram

Figure 55. Intel FPGA I²C to Avalon-MM Host Bridge Block Diagram



16.2.2. N-byte Addressing

This IP supports up to a 4 bytes addressing mode. You can select which byte addressing mode you want to use in Platform Designer.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

The Avalon Address width present at the Avalon host interface is fixed at 32 bits. If you select other than a 4 bytes addressing mode, zeros are added to the most significant bit(s) (MSB) of the Avalon Address width. For example in 2 bytes addressing mode, only the lower 16 bits of the address width are used while the upper 16 bits are zero.

- When byte addressing mode = 1, address width in use = 8 + address stealing bit
- When byte addressing mode = 2, address width in use = 16 + address stealing bit
- When byte addressing mode = 3, address width in use = 24 + address stealing bit
- When byte addressing mode = 4, address width in use = 32

There is an address counter inside the I²C to Avalon host interface translator block. The counter rolls over at the maximum upper address bound according to the byte addressing mode plus one address stealing bit. It does not continue incrementing up the full address range of the Avalon address size. For example, the address counter rolls over at 128 K memory size in 2 bytes addressing mode plus one address stealing bit.

16.2.3. N-byte Addressing with N-bit Address Stealing

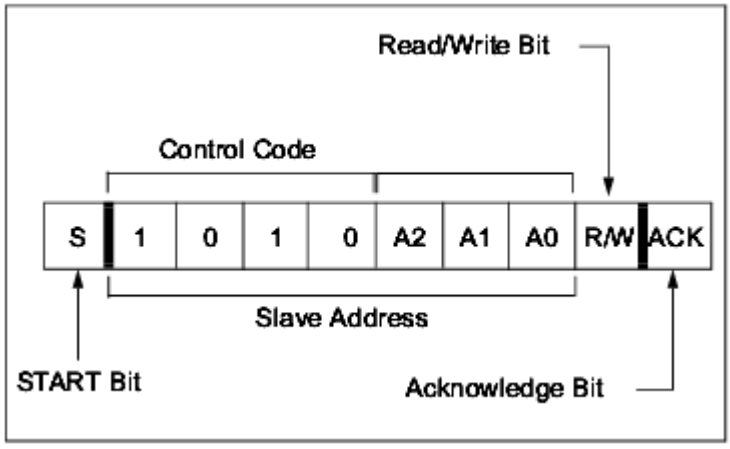
This IP supports up to 3-bit address stealing. In Platform Designer, you can configure which address stealing mode to use. The address stealing bits (A0, A1, A2) are added into the second, third, and forth bits of the control byte to expand the contiguous address space. If no address stealing bits are used, then the second, third, and forth bits of the control byte are used as agent address bits.

Note: When in 3-bit address stealing mode, you must make sure the three least significant bits (LSB) of the agent address are zero.

The maximum upper bound of the internal address counter is:

BYTEADDRWIDTH + address stealing bit(s)

Figure 56. 8-bit Control Byte Example



16.2.4. Read Operation

The Avalon read data width is 32 bits wide. A 32-bit width limits the bridge to only issue word align Avalon addresses. It also allows the upstream I²C host to read any sequence of bytes on any address alignment. The conversion logic which sits between the Avalon interface and I²C interface, translates the address alignment and returns the correct 8-bit data to the I²C host from the 32-bit Avalon read data.

Read Operation conversion logic flow:

- Checks the address alignment issued by the I²C host (first byte, second byte, third byte or forth byte).
- Issues a word align Avalon address according to the address sent by the I²C host with the two LSBs zero.
- Returns read data to the I²C host according to the address alignment.

This IP supports three types of read operations:

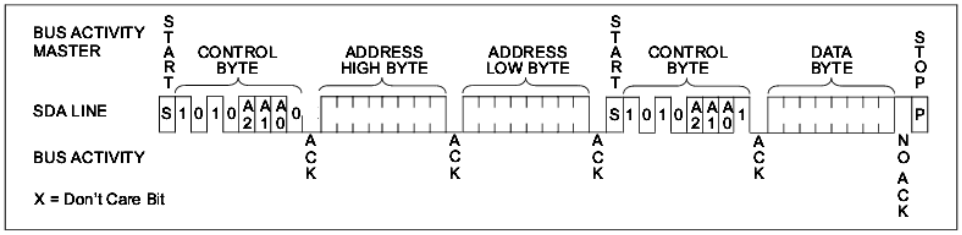
- Random address read
- Current address read
- Sequential read

Upon receiving of the agent address with the R/W bit set to one, the bridge issues an acknowledge to the I²C host. The bridge keeps the Avalon read signal high for one clock cycle with the Avalon wait request signal low, then receives an 8-bit Avalon read data word and upstreams the read data to the I²C host.

16.2.4.1. Random Address Read

Random read operations allow the upstream I²C host to access any memory location in a random manner. To perform this type of read operation, you must first set the byte address. The I²C host issues a byte address to the bridge as part of a write operation then followed by a read command. After the read command is issued, the internal address counter points to the address location following the one that was just read. The upper address bits are transferred first, followed by the LSB(s).

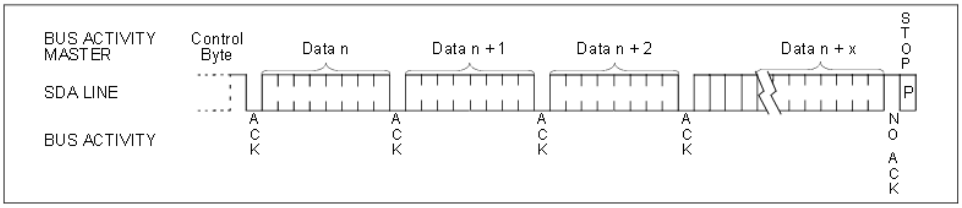
Figure 57. Random Address Read



16.2.4.2. Sequential Address Read

Sequential reads are initiated in the same way as a random read except after the bridge has received the first data byte, the upstream I²C host issues an acknowledge as opposed to a Stop condition. This directs the bridge to keep the Avalon read signal high for the next sequential address. The internal address counter increments by one at the completion of each read operation and allows the entire memory contents to be serially read during one operation.

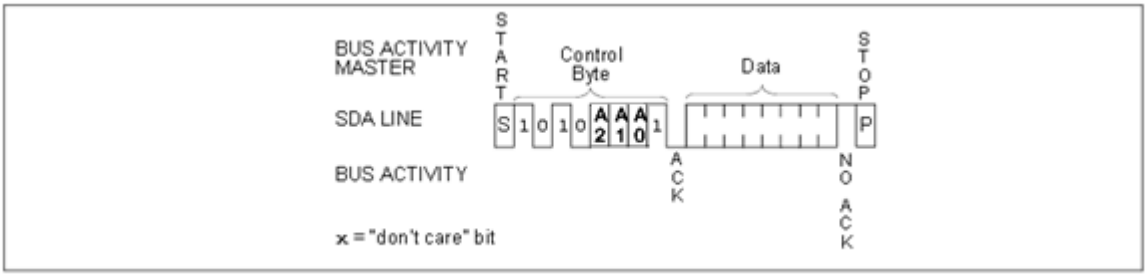
Figure 58. Sequential Address Read



16.2.4.3. Current Address Read

This IP contains an internal address counter that maintains the address of the last word accessed incremented by one. Therefore, if the previous access was to address n, the next current address read operation would access data from address n + 1.

Figure 59. Current Address Read



16.2.5. Write Operation

The Avalon write data width interface is 32 bits wide. A 32-bit width limits the bridge to only issue word align Avalon addresses. It also allows the upstream I²C host to write to any sequence of bytes on any address alignment. There is a conversion logic which sits between the Avalon interface and the I²C interface.

Write operation conversion logic flow:

- Checks the address alignment issued by the I²C host.
- Enables data by setting `byteenable` high to indicate which byte address the I²C host wants to write into.

  Note: If the address issued by I²C host is `0x03h`, the `byteenable` is 4'b1000.
- Combines multiple bytes of data into a 32-bit packet if their addresses are sequential.

  Note: If the first write is to address `0x04` and the second write is to address `0x05`, then `byteenable` is 4'b0011.

Legal `byteenable` combinations are 4'b0001, 4'b0010, 4'b0100, 4'b1000, 4'b0011, 4'b1100 and 4'b1111.

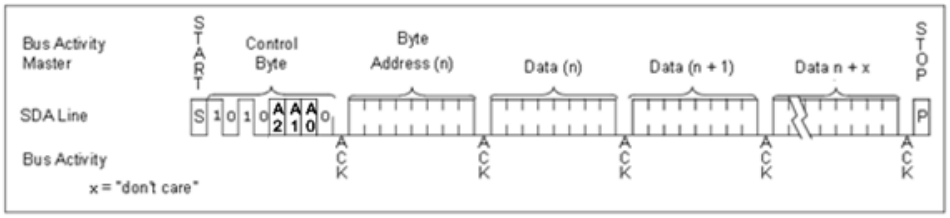
- If the write request issued by the I²C host ends up with an illegal `byteenable` combination such as, 4'b0110, 4'b0111, or 4'b1110, then the bridge generates multiple Avalon byte writes.

  Note: If the sequential write request from the I²C host starts from `0x0` and ends at `0x02` (illegal `byteenable`, b'0111), then the bridge will generate three Avalon write requests with legal `byteenable` 4'b0001, 4'b0010 and 4'b0100.

- Issues a word align Avalon address according to the address sent by the I²C host with the two LSB set to zero.

Upon receiving of the agent address with the R/W bit set to zero, the bridge issues an acknowledge to the I²C host. The next byte transmitted by the host is the byte address. The byte address is written into the address counter inside the bridge. The bridge acknowledges the I²C host again and the host transmits the data byte to be written into the addressed memory location. The host keeps sending data bytes to the bridge and terminates the operation with a Stop condition at the end.

Figure 60. Write Operation



16.2.6. Interacting with Multi-Hosts

This IP core is able to integrate multiple I²C hosts provided the I²C hosts support the arbitration feature. The hosts which support arbitration always compares the data value it drives into the bus with the actual value observed on the bus. If both sets of data do not match, then the host loses arbitration. The I²C agent core in the bridge does not observe the bus.

For example, let's say Host 1 is writing to the bridge while Host 2 is performing a read. Host 1 will win the arbitration during the R/W bit because Host 1 is pulling down the bus, while Host 2 is driving high to the bus. The effective value of the bus during the R/W bit cycle is zero. In this case, Host 2 knows it loses the arbitration because it observes a different value on the bus than what is being driven.

If both hosts are performing a write, then the arbitration process checks the different data value driven by both hosts to determine which one wins the arbitration.

If both hosts are performing a read, then no one loses the arbitration since the single agent is driving the bus to both hosts (no collision).

16.3. Platform Designer Parameters

Figure 61. Intel FPGA I²C Agent to Avalon MM Host Bridge Interface

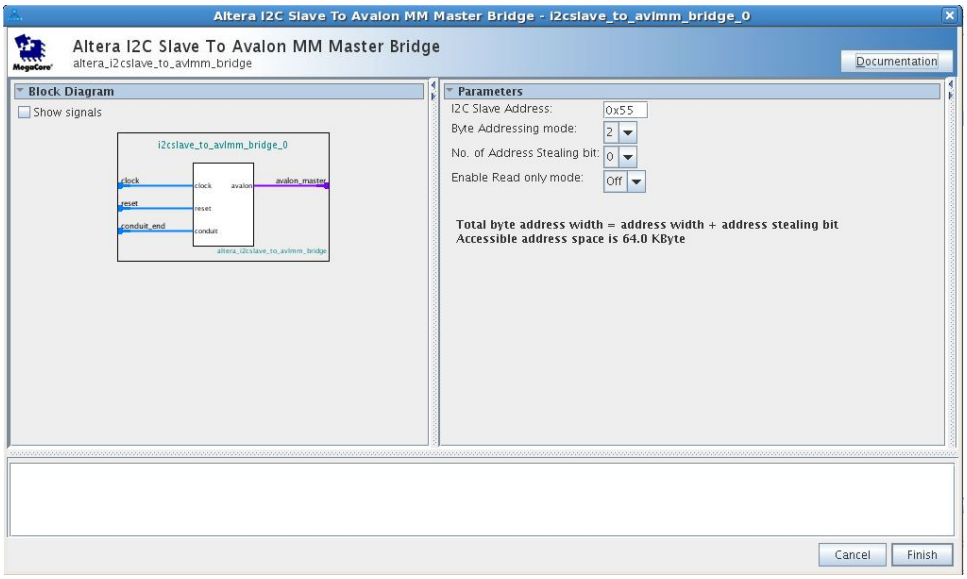


Table 147. Platform Designer Parameters

Parameter	Legal Values	Default Values	Description
I²C Agent Address	0:127	127	This parameter represents the target address of the I²C agent which sits in the bridge.
Byte Addressing mode	1, 2, 3, 4	2	This parameter allows you to select the number of address bytes you want to configure according to the flash capacity used. • 1=8 address bit • 2=16 address bit • 3=24 address bit • 4=32 address bit
Number of Address Stealing bit	0, 1, 2, 3	0	This parameter allows you to select the number of address stealing bits to expand the contiguous address space.
Enable Read only mode	ON, OFF	OFF	Enables read only support where the write operation is removed to improve resource count.

16.4. Signals

Table 148. Intel FPGA I²C Agent to Avalon-MM Host Bridge Signals

Signal	Width	Direction	Description
Avalon-MM interface			
clk	1	Input	Clock signal
rst_n	1	Input	Reset signal

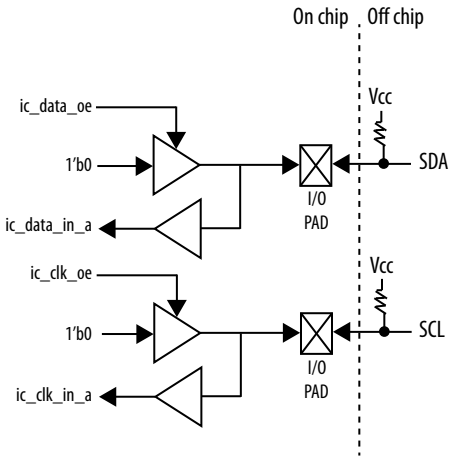
continued...

Signal	Width	Direction	Description
address	32	Output	Avalon-MM address bus. The address bus is in word addressing.
byteenable	4	Output	Avalon-MM byteenable signal.
read	1	Output	Avalon-MM read request signal.
readdata	32	Input	Avalon-MM read data bus.
readdatavalid	1	Input	Avalon-MM read data valid signal.
waitrequest	1	Input	Avalon-MM wait request signal
write	1	Output	Avalon-MM write request signal.
writedata	32	Output	Avalon-MM write data bus.
Serial conduit interface for I²C			
i2c_data_in	1	Input	I²C agent conduit data input signal.
i2c_clk_in	1	Input	I²C agent conduit clock input signal.
i2c_data_oe	1	Output	I²C agent conduit data output signal.
i2c_clk_oe	1	Output	I²C agent conduit clock output signal.

16.5. How to Translate the Bridge's I²C Data and I²C I/O Ports to an I²C Interface

In order to translate the bridges I²C data and I²C I/O ports to an I²C interface refer to the figure below. You need to connect a tri-state buffer to the cores I²C data and clock related ports to form SDA and SCL.

Figure 62.



Example 9. Translating the Bridge's I²C Data and I²C I/O Ports to an I²C Interface

```
module top (
    inout  tri1         fx2_scl,
    inout  tri1         fx2_sda

);

wire fx2_sda_in;
wire fx2_scl_in;
wire fx2_sda_oe;
wire fx2_scl_oe;

assign fx2_scl_in = fx2_scl;
assign fx2_sda_in = fx2_sda;
assign fx2_scl = fx2_scl_oe ? 1'b0 : 1'bz;
assign fx2_sda = fx2_sda_oe ? 1'b0 : 1'bz;

proj_1 u0 (
        .i2cslave_to_avlmm_bridge_0_conduit_end_conduit_data_in
(fx2_sda_in), // i2c_bridge.conduit_data_in
        .i2cslave_to_avlmm_bridge_0_conduit_end_conduit_clk_in
(fx2_scl_in), //           .conduit_clk_in
        .i2cslave_to_avlmm_bridge_0_conduit_end_conduit_data_oe
(fx2_sda_oe), //           .conduit_data_oe
        .i2cslave_to_avlmm_bridge_0_conduit_end_conduit_clk_oe
(fx2_scl_oe)  //           .conduit_clk_oe
    );

endmodule
```

16.6. Intel FPGA I²C Agent to Avalon-MM Host Bridge Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2021.03.29	21.1	Added support for Intel eASIC N5X devices.
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
October 2016	2016.10.28	Updates: • Table 148 on page 202 — `address` direction updated — `waitrequest` added
June 2016	2016.06.17	New topic: • How to Translate the Bridge's I2C Data and I2C I/O Ports to an I2C Interface on page 204
May 2016	2016.05.03	Initial release.

17. Intel FPGA Avalon Compact Flash Core

Attention: The Intel FPGA Avalon Compact Flash Core is part of a product obsolescence and support discontinuation schedule. For new design, Intel recommends that you use other IPs with equivalent functions. To see a list of available IPs, refer to the Intel FPGA IP Portfolio page.

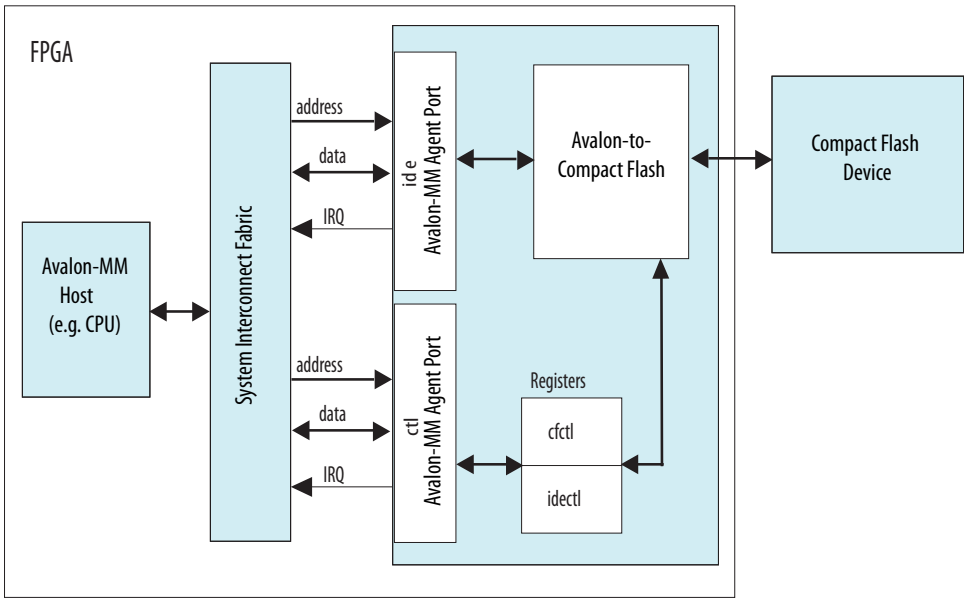
17.1. Core Overview

The Intel FPGA Avalon Compact Flash core allows you to connect systems built on Platform Designer to CompactFlash storage cards in true IDE mode by providing an Avalon Memory-Mapped (Avalon-MM) interface to the registers on the storage cards. The core supports PIO mode 0.

The Compact Flash core also provides an Avalon-MM agent interface which can be used by Avalon-MM host peripherals such as a Nios II processor to communicate with the Compact Flash core and manage its operations.

17.2. Functional Description

Figure 63. System With a CompactFlash Core



As shown in the block diagram, the Compact Flash core provides two Avalon-MM agent interfaces: the `ide` agent port for accessing the registers on the CompactFlash device and the `ctl` agent port for accessing the core's internal registers. These registers can

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

be used by Avalon-MM host peripherals such as a Nios II processor to control the operations of the Compact Flash core and to transfer data to and from the CompactFlash device.

You can set the Compact Flash core to generate two active-high interrupt requests (IRQs): one signals the insertion and removal of a CompactFlash device and the other passes interrupt signals from the CompactFlash device.

The Compact Flash core maps the Avalon-MM bus signals to the CompactFlash device with proper timing, thus allowing Avalon-MM host peripherals to directly access the registers on the CompactFlash device.

For more information, refer to the CF+ and CompactFlash specifications available at CompactFlash Association website.

17.3. Required Connections

The table below lists the required connections between the Compact Flash core and the CompactFlash device.

Table 149. Core to Device Required Connections

CompactFlash Interface Signal Name	Pin Type	CompactFlash Pin Number
addr[0]	Output	20
addr[1]	Output	19
addr[2]	Output	18
addr[3]	Output	17
addr[4]	Output	16
addr[5]	Output	15
addr[6]	Output	14
addr[7]	Output	12
addr[8]	Output	11
addr[9]	Output	10
addr[10]	Output	8
atasel_n	Output	9
cs_n[0]	Output	7
cs_n[1]	Output	32
data[0]	Input/Output	21
data[1]	Input/Output	22
data[2]	Input/Output	23
data[3]	Input/Output	2
data[4]	Input/Output	3
data[5]	Input/Output	4

continued...

CompactFlash Interface Signal Name	Pin Type	CompactFlash Pin Number
data[6]	Input/Output	5
data[7]	Input/Output	6
data[8]	Input/Output	47
data[9]	Input/Output	48
data[10]	Input/Output	49
data[11]	Input/Output	27
data[12]	Input/Output	28
data[13]	Input/Output	29
data[14]	Input/Output	30
data[15]	Input/Output	31
detect	Input	25 or 26
intrq	Input	37
iord_n	Output	34
iordy	Input	42
iowr_n	Output	35
power	Output	CompactFlash power controller, if present
reset_n	Output	41
rfu	Output	44
we_n	Output	46

17.4. Software Programming Model

This section describes the software programming model for the Compact Flash core.

17.4.1. HAL System Library Support

The Intel-provided HAL API functions include a device driver that you can use to initialize the Compact Flash core. To perform other operations, use the low-level macros provided.

For more information on the macros, refer to the "Software Files" section.

Related Information

Software Files on page 208

17.4.2. Software Files

The Compact Flash core provides the following software files. These files define the low-level access to the hardware. Application developers should not modify these files.

- `altera_avalon_cf_regs.h`—The header file that defines the core's register maps.
- `altera_avalon_cf.h, altera_avalon_cf.c`—The header and source code for the functions and variables required to integrate the driver into the HAL system library.

17.4.3. Register Maps

This section describes the register maps for the Avalon-MM agent interfaces.

17.4.3.1. Ide Registers

The `ide` port in the Compact Flash core allows you to access the IDE registers on a CompactFlash device.

Table 150. Ide Register Map

Offset	Register Names	
	Read Operation	Write Operation
0	RD Data	WR Data
1	Error	Features
2	Sector Count	Sector Count
3	Sector No	Sector No
4	Cylinder Low	Cylinder Low
5	Cylinder High	Cylinder High
6	Select Card/Head	Select Card/Head
7	Status	Command
14	Alternate Status	Device Control

17.4.3.2. Ctl Registers

The `ctl` port in the Compact Flash core provides access to the registers which control the core's operation and interface.

Table 151. Ctl Register Map

Offset	Register	Fields				
		31:4	3	2	1	0
0	cfctl	Reserved	IDET	RST	PWR	DET
1	idectl	Reserved				IIDE
2	Reserved	Reserved				
3	Reserved	Reserved				

17.4.3.3. Cfctl Register

The cfctl register controls the operations of the Compact Flash core. Reading the cfctl register clears the interrupt.

Table 152. cfctl Register Bits

Bit Number	Bit Name	Read/Write	Description
0	DET	RO	Detect. This bit is set to 1 when the core detects a CompactFlash device.
1	PWR	RW	Power. When this bit is set to 1, power is being supplied to the CompactFlash device.
2	RST	RW	Reset. When this bit is set to 1, the CompactFlash device is held in a reset state. Setting this bit to 0 returns the device to its active state.
3	IDET	RW	Detect Interrupt Enable. When this bit is set to 1, the Compact Flash core generates an interrupt each time the value of the `det` bit changes.

17.4.3.4. idectl Register

The `idectl` register controls the interface to the CompactFlash device.

Table 153. idectl Register

Bit Number	Bit Name	Read/Write	Description
0	IIDE	RW	IDE Interrupt Enable. When this bit is set to 1, the Compact Flash core generates an interrupt following an interrupt generated by the CompactFlash device. Setting this bit to 0 disables the IDE interrupt.

17.5. Intel FPGA Avalon Compact Flash Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2021.03.29	21.1	Added product obsolescence and support discontinuation notice.
2018.05.07	18.0	Corrected the register name for offset 14 of *Ide Registers*.

Date	Version	Changes
July 2014	2014.07.24	Removed mention of SOPC Builder, updated to Platform Designer
December 2010	v10.1.0	Removed the "Device Support", "Instantiating the Core in SOPC Builder", and "Referenced Documents" sections.
July 2010	v10.0.0	No change from previous release.
November 2009	v9.1.0	No change from previous release.
March 2009	v9.0.0	No change from previous release.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	v8.0.0	Added the mode supported by the Compact Flash core.

18. EPCS/EPCQA Serial Flash Controller Core

18.1. Core Overview

The EPCS/EPCQA serial flash controller core with Avalon interface allows Nios II systems to access an Intel EPCS/EPCQA serial configuration device. Intel provides drivers that integrate into the Nios II hardware abstraction layer (HAL) system library, allowing you to read and write the EPCS/EPCQA device using the familiar HAL application program interface (API) for flash devices.

Note: The EPCS/EPCQA serial flash controller core is not supported by Intel Cyclone 10 LP devices.

Using the EPCS/EPCQA serial flash controller core, Nios II systems can:

- Store program code in the EPCS/EPCQA device. The EPCS/EPCQA serial flash controller core provides a boot-loader feature that allows Nios II systems to store the main program code in an EPCS/EPCQA device.
- Store non-volatile program data, such as a serial number, a NIC number, and other persistent data.
- Manage the device configuration data. For example, a network-enabled embedded system can receive new FPGA configuration data over a network, and use the core to program the new data into an EPCS/EPCQA serial configuration device.

  The EPCS/EPCQA serial flash controller core is Platform Designer-ready and integrates easily into any Platform Designer-generated system. The flash programmer utility in the Nios II IDE allows you to manage and program data contents into the EPCS/EPCQA device.

  Note: Intel is offering an alternative device for EPCS. Refer to *Product Discontinuance Notification PDN1708* for more details. For information about migration, refer to *AN822: Intel Configuration Device Migration Guideline*.

  For information about the EPCS/EPCQA serial configuration device family, refer to the *Serial Configuration Devices Data Sheet*.

  For details about using the Nios II HAL API to read and write flash memory, refer to the *Nios II Software Developer's Handbook*.

  For details about managing and programming the EPCS/EPCQA memory contents, refer to the *Nios II Flash Programmer User Guide*.

  For Nios II processor users, the EPCS/EPCQA serial flash controller core supersedes the Active Serial Memory Interface (ASMI) device. New designs should use the EPCS/EPCQA serial flash controller core instead of the ASMI core.

Related Information

- Serial Configuration Devices (EPCS1, EPCS4, EPCS16, EPCS64 and EPCS128) Data Sheet

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

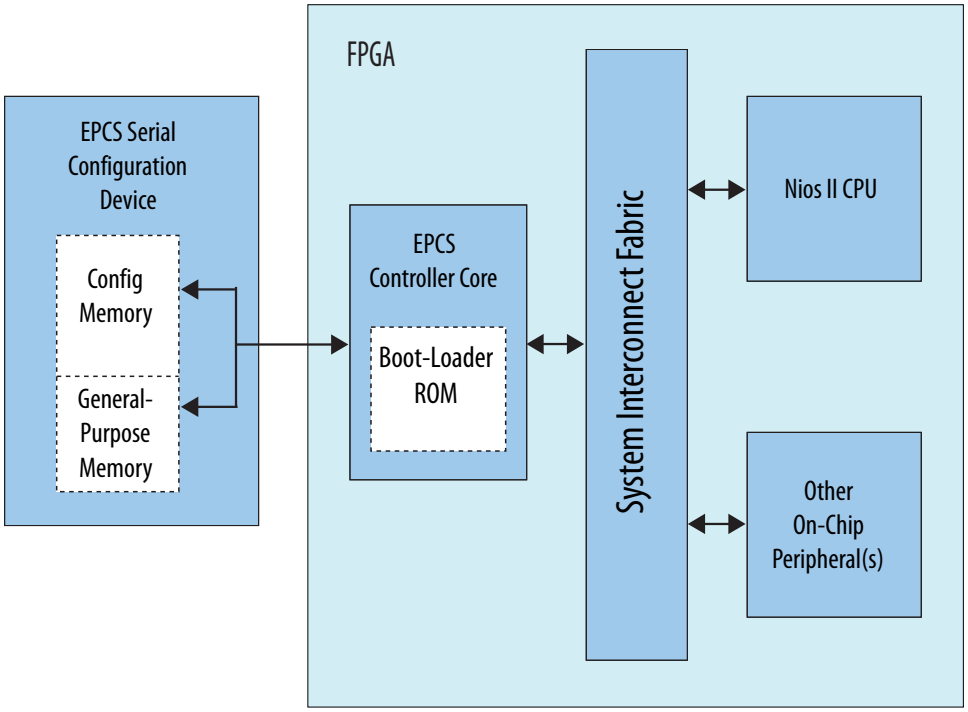
- Nios II Classic Software Developer's Handbook
- Nios II Flash Programmer User Guide
- Intel Configuration Device Migration Guideline
- Product Discontinuance Notification PDN1708

18.2. Functional Description

As shown below, the EPCS/EPCQA device's memory can be thought of as two separate regions:

- **FPGA configuration memory**—FPGA configuration data is stored in this region.
- **General-purpose memory**—If the FPGA configuration data does not fill up the entire EPCS/EPCQA device, any left-over space can be used for general-purpose data and system startup code.

Figure 64. Nios II System Integrating an EPCS/EPCQA Serial Flash Controller Core



- By virtue of the HAL generic device model for flash devices, accessing the EPCS/EPCQA device using the HAL API is the same as accessing any flash memory. The EPCS/EPCQA device has a special-purpose hardware interface, so Nios II programs must read and write the EPCS/EPCQA memory using the provided HAL flash drivers.

The EPCS/EPCQA serial flash controller core contains an on-chip memory for storing a boot-loader program. When used in conjunction with Cyclone and Cyclone II devices, the core requires 512 bytes of boot-loader ROM. For Cyclone III, Cyclone IV, Intel Cyclone 10 LP, Stratix II, and newer device families in the Stratix series, the core requires 1 KByte of boot-loader ROM. The Nios II processor can be configured to boot from the EPCS/EPCQA serial flash controller core. To do so, set the Nios II reset

address to the base address of the EPCS/EPCQA serial flash controller core. In this case, after reset the CPU first executes code from the boot-loader ROM, which copies data from the EPCS/EPCQA general-purpose memory region into a RAM. Then, program control transfers to the RAM. The Nios II IDE provides facilities to compile a program for storage in the EPCS/EPCQA device, and create a programming file to program into the EPCS/EPCQA device.

For more information, refer to the *Nios II Flash Programmer User Guide*.

If you program the EPCS/EPCQA device using the Intel Quartus Prime Programmer, all previous content is erased. To program the EPCS/EPCQA device with a combination of FPGA configuration data and Nios II program data, use the Nios II IDE flash programmer utility.

The Intel EPCS/EPCQA configuration device connects to the FPGA through dedicated pins on the FPGA, not through general-purpose I/O pins. In all Intel device families except Cyclone III, Cyclone IV, and Intel Cyclone 10 LP the EPCS/EPCQA serial flash controller core does not create any I/O ports on the top-level Platform Designer system module. If the EPCS/EPCQA device and the FPGA are wired together on a board for configuration using the EPCS/EPCQA device (in other words, active serial configuration mode), no further connection is necessary between the EPCS/EPCQA serial flash controller core and the EPCS/EPCQA device. When you compile the Platform Designer system in the Intel Quartus Prime software, the EPCS/EPCQA serial flash controller core signals are routed automatically to the device pins for the EPCS/EPCQA device.

You, however, have the option not to use the dedicated pins on the FPGA (active serial configuration mode) by turning off the respective parameters in the MegaWizard interface. When this option is turned off or when the target device is a Cyclone III, Cyclone IV device, or Intel Cyclone 10 LP you have the flexibility to connect the output pins, which are exported to the top-level design, to any EPCS/EPCQA devices. Perform the following tasks in the Intel Quartus Prime software to make the necessary pin assignments:

- On the **Dual-purpose pins** page (**Assignments** > **Devices** > **Device and Pin Options**), ensure that the following pins are assigned to the respective values:
  - `Data[0]` = **Use as regular I/O**
  - `Data[1]` = **Use as regular I/O**
  - `DCLK` = **Use as regular I/O**
  - `FLASH_nCE/nCS0` = **Use as regular I/O**
- Using the Pin Planner (**Assignments > Pins**), ensure that the following pins are assigned to the respective configuration functions on the device:
  - `data0_to_the_epcs_controller` = `DATA0`
  - `sdo_from the_epcs_controller` = `DATA1,ASDO`
  - `dclk_from_epcs_controller` = `DCLK`
  - `sce_from_the_epcs_controller` = `FLASH_nCE`

Related Information

Nios II Flash Programmer User Guide

18.2.1. Avalon-MM Agent Interface and Registers

The EPCS/EPCQA serial flash controller core has a single Avalon-MM agent interface that provides access to both boot-loader code and registers that control the core. As shown in below, the first segment is dedicated to the boot-loader code, and the next seven words are control and data registers. A Nios II CPU can read the instruction words, starting from the core's base address as flat memory space, which enables the CPU to reset the core's address space.

The EPCS/EPCQA serial flash controller core includes an interrupt signal that can be used to interrupt the CPU when a transfer has completed.

Table 154. EPCS/EPCQA Serial Flash Controller Core Register Map

Offset (32-bit Word Address)	Register Name	R/W	Bit Description
			31:0
0x00 .. 0xFF	Boot ROM Memory	R	Boot Loader Code
0x100	Read Data	R	
0x101	Write Data	W	
0x102	Status	R/W	
0x103	Control	R/W	
0x104	Reserved	—	
0x105	Slave Enable	R/W	
0x106	End of Packet	R/W	

Note: Intel does not publish the usage of the control and data registers. To access the EPCS/EPCQA device, you must use the HAL drivers provided by Intel.

18.3. Configuration

The core must be connected to a Nios II processor. The core provides drivers for HAL-based Nios II systems, and the precompiled boot loader code compatible with the Nios II processor.

In device families other than Cyclone III, Cyclone IV, and Intel Cyclone 10 LP, you can use the MegaWizard™ interface to configure the core to use general I/O pins instead of dedicated pins by turning off both parameters, **Automatically select dedicated active serial interface**, if supported and **Use dedicated active serial interface**.

Only one EPCS/EPCQA serial flash controller core can be instantiated in each FPGA design.

18.4. Interface Signals

Table 155. Interface Signals

Signal Name	Width	Direction	Description
Clock Interface			
clk	1	Input	Input clock used to clock the IP core.
Reset Interface			
reset_n	1	Input	Synchronous reset used to reset the IP core.
resetrequest	1	Input	Reset RAM memory.
Avalon Memory-Mapped Interface Agent Interface			
address	9	Input	Avalon Memory-Mapped Interface address lines from the system interconnect fabric.
chipselect	1	Input	Avalon Memory-Mapped Interface chip-select signal.
dataavailable	1	Output	Flow control signal which indicates that the data is ready for read transfer.
endofpacket	1	Output	Flow control signal which indicates the end of SPI data word.
read_n	1	Input	Avalon Memory-Mapped Interface read request signal.
readdata	32	Output	Avalon Memory-Mapped Interface read data to system interconnect fabric for read transfer.
readyfordata	1	Output	Flow control signal which indicates the availability to accept a write transfer.
write_n	1	Input	Avalon Memory-Mapped Interface write request signal.
write_data	32	Input	Avalon Memory-Mapped Interface write data from the system interconnect fabric.
Interrupt Signal			
irq	1	Output	Interrupt signal.
Conduit Ports			
dclk	1	Output	Provides a clock signal to the flash device.
sce	1	Output	Provides the ncs signal to the flash device.
sdo	1	Output	Output port to feed data into flash device.
data0	1	Input	Input port to feed data from flash device.

18.5. Software Programming Model

This section describes the software programming model for the EPCS/EPCQA serial flash controller core. Intel provides HAL system library drivers that enable you to erase and write the EPCS/EPCQA memory using the HAL API functions. Intel does not publish the usage of the cores registers. Therefore, you must use the HAL drivers provided by Intel to access the EPCS/EPCQA device.

Related Information

Using Flash Devices

18.5.1. HAL System Library Support

The Intel-provided driver implements a HAL flash device driver that integrates into the HAL system library for Nios II systems. Programs call the familiar HAL API functions to program the EPCS/EPCQA memory. You do not need to know the details of the underlying drivers to use them.

The driver for the EPCS/EPCQA device is excluded when the reduced device drivers option is enabled in a BSP or system library. To force inclusion of the EPCS/EPCQA drivers in a BSP with the reduced device drivers option enabled, you can define the preprocessor symbol, ALT_USE_EPCS/EPCQA_FLASH, before including the header, as follows:

```
#define ALT_USE_EPCS/EPCQA_FLASH

#include <altera_avalon_epcs_flash_controller.h>
```

The HAL API for programming flash, including C-code examples, is described in detail in the Nios II Classic Software Developer's Handbook.

For details about managing and programming the EPCS/EPCQA device contents, refer to the Nios II Flash Programmer User Guide.

18.5.2. Software Files

The EPCS/EPCQA serial flash controller core provides the following software files. These files provide low-level access to the hardware and drivers that integrate into the Nios II HAL system library. Application developers should not modify these files.

- `altera_avalon_epcs_flash_controller.h`, `altera_avalon_epcs_flash_controller.c`—Header and source files that define the drivers required for integration into the HAL system library.
- `epcs_commands.h`, `epcs_commands.c`—Header and source files that directly control the EPCS/EPCQA device hardware to read and write the device. These files also rely on the Intel FPGA SPI core drivers.

18.6. Driver API

Table 156. alt_epcs_flash_get_info

Prototype:	alt_epcs_flash_get_info(alt_flash_fd* fd, flash_region** info, int* number_of_regions)
Include:	<altera_avalon_epcs_flash_controller.h>
Parameter:	• fd – pointer to general flash device structure • info- pointer to the flash region • number_of_regions- pointer to the number of regions
Return:	Return 0 if successful and otherwise return: • -ENOMEM for number of region more than maximum number of flash region • -EIO for possibly hardware problem
Description:	Pass the table of erase blocks to the user.

Table 157. alt_epcs_flash_erase_block

Prototype:	alt_epcs_flash_erase_block(alt_flash_dev* flash_info, int block_offset)
Include:	<altera_avalon_epcs_flash_controller.h>
Parameter:	• flash_info – pointer to flash device structure • block_offset- byte-addressed offset, from start of flash of the sector to be erased
Return:	Return 0 if successful.
Description:	Erase the selected erase block

Table 158. alt_epcs_flash_write_block

Prototype:	alt_epcs_flash_write_block(alt_flash_dev* flash_info, int block_offset, int data_offset, const void* data, int length)
Include:	<altera_avalon_epcs_flash_controller.h>
Parameter:	• flash_info – pointer to flash device structure • block_offset- the base of the current erase block • data_offset – absolute address of the beginning of the write-destination • data – data to be written • length – size of writing
Return:	Return 0 if successful.
Description:	Write one block/sector of data to the flash. The length of the write cannot spill into the adjacent sector.

Table 159. alt_epcs_flash_write

Prototype:	alt_epcs_flash_write (alt_flash_dev* flash_info, int offset, const void* src_addr, int length)
Include:	<altera_avalon_epcs_flash_controller.h>
Parameter:	• flash_info – pointer to flash device structure • offset- byte offset (unaligned access) of write to flash memory • src_addr – source buffer • length – size of writing
Return:	Return 0 if successful.
Description:	Program the data into the flash at the selected address. This function automatically erases a block as needed.

Table 160. alt_epcs_flash_read

Prototype:	alt_epcs_flash_read(alt_flash_dev* flash_info, int offset, void* dest_addr, int length)
Include:	<altera_avalon_epcs_flash_controller.h>
Parameter:	• flash_info – pointer to flash device structure • offset- offset read from flash memory • dest_addr – destination buffer • length – size of writing
Return:	Return 0 if successful.
Description:	Read data from flash at the selected address.

18.7. EPCS/EPCQA Serial Flash Controller Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.09.24	18.1	Added new sections: *Interface Signals* and *Driver API*.
2018.05.07	18.0	• Added information about discontinuance of EPCS devices in *Core Overview*. • Changed instances of EPCS to EPCS/EPCQA.

Date	Version	Changes
July 2014	2014.07.24	Removed mention of SOPC Builder, updated to Platform Designer
December 2013	v13.1.0	Removed Cyclone and Cyclone II device information in the "EPCS Serial Flash Controller Core Register Map" table.
December 2010	v10.1.0	Removed the “Device Support”, “Instantiating the Core in SOPC Builder”, and “Referenced Documents” sections.
July 2010	v10.0.0	No change from previous release.
November 2009	v9.1.0	Revised descriptions of register fields and bits. Updated the section on HAL System Library Support.
March 2009	v9.0.0	Updated the boot ROM memory offset for other device families in the EPCS Serial Flash Controller Core Register Map" table.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	v8.0.0	Updated the boot rom size. Added additional steps to perform to connect output pins in Cyclone III devices.

19. Intel FPGA Serial Flash Controller Core

Intel FPGA Serial Flash Controller Core wraps around the ASMI Parallel IP and the logic that converts the ASMI parallel conduit interface to Avalon interface. This IP core supports the F_{MAX} of 25MHz.

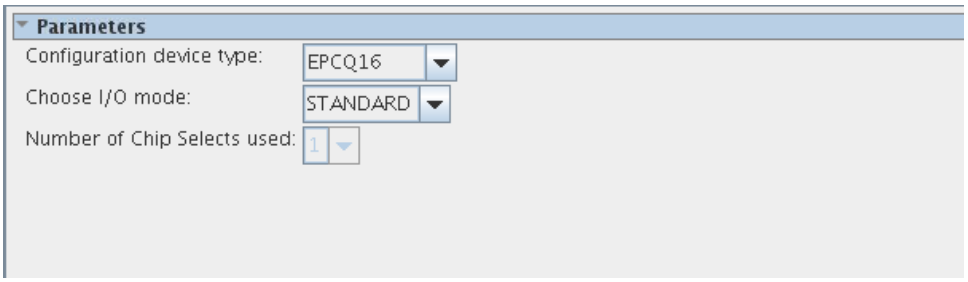
Note: The Intel FPGA Serial Flash Controller core does not support Intel Stratix 10 and Intel Agilex devices.

Related Information

Generic Serial Flash Interface Intel FPGA IP Core User Guide
For your new (non Nios II) designs, Intel recommends you use the Generic Serial Flash Interface Intel FPGA IP core rather than this IP core.

19.1. Parameters

Figure 65. Platform Designer Parameters



19.1.1. Configuration Device Types

The following device types can be selected through the configuration device type drop down menu. Here you can specify the EPCQ type you want to use.

- EPCS16
- EPCS64
- EPCS128
- EPCQ16
- EPCQ32
- EPCQ64
- EPCQ128
- EPCQ256
- EPCQ512
- EPCQL256

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

- EPCQL512
- EPCQL1024
- EPCQ4A
- EPCQ16A
- EPCQ32A
- EPCQ64A
- EPCQ128A

While using EPCS, EPCQ, or EPCQL devices for Intel FPGA Serial Flash Controller, you must set MSEL pins for AS or ASx4 configuration scheme.

19.1.2. I/O Mode

From the parameters menu you can select either standard (AS configuration) or Quad (ASx4 configuration) I/O mode.

19.1.3. Chip Selects

For Intel Arria 10 devices, you can select up to three flash chips from the parameters menu.

19.1.4. Interface Signals

Table 161. Interface Signals

Signal	Width	Direction	Description
Clock			
clk	1	Input	25MHz maximum input clock.
Reset			
reset_n	1	Input	Asynchronous reset used to reset Quad SPI Controller
Avalon-MM Agent Interface for CSR (avl_csr)			
avl_csr_addr	3	Input	Avalon-MM address bus. The address bus is in word addressing.
avl_csr_read	1	Input	Avalon-MM read control to csr
avl_csr_write	1	Input	Avalon-MM write control to csr
avl_csr_waitrequest	1	Output	Avalon-MM waitrequest control from csr
avl_csr_wrdata	32	Input	Avalon-MM write data bus to csr
avl_csr_rddata	32	Output	Avalon-MM read data bus from csr

continued...

Signal	Width	Direction	Description
avl_csr_rddata_valid	1	Output	Avalon-MM read data valid which indicates that csr read data is available
Interrupt Signals			
irq	1	Output	Interrupt signal to determine if there is an illegal write or illegal erase
Avalon-MM Agent Interface for Memory Access (avl_ mem)			
avl_mem_addr	*	Input	Avalon-MM address bus. The address bus is in word addressing. The width of the address will depends on the flash memory density minus 2. If you are using Intel Arria 10, then the MSB bits will be used for chip select information. User is allowed to select the number of chip select needed in the GUI. If user selects 1 chip select, there will be no extra bit added to avl_mem_addr. If user select 2 chip selects, there will be one extra bit added to avl_mem_addr. Chip 1 – b'0 Chip 2 – b'1 If user select 3 chip selects, there will be two extra bit added to avl_mem_addr. Chip 1 – b'00 Chip 2 – b'01 Chip 3 – b'10
avl_mem_read	1	Input	Avalon-MM read control to memory
avl_mem_write	1	Input	Avalon-MM write control to memory
avl_mem_wrdata	32	Input	Avalon-MM write data bus to memory
avl_mem_byteenble	4	Input	Avalon-MM write data enable bit to memory. During bursting mode, byteenable bus bit will be all high always, 4'b1111.
avl_mem_burstcount	7	Input	Avalon-MM burst count for memory. Value range from 1 to 64
avl_mem_waitrequest	1	Output	Avalon-MM waitrequest control from memory
avl_mem_rddata	32	Output	Avalon-MM read data bus from memory
avl_mem_rddata_valid	1	Output	Avalon-MM read data valid which indicates that memory read data is available

continued...

Signal	Width	Direction	Description
Conduit Interface			
flash_dataout	4	Input/Output	Input/output port to feed data from flash device
flash_dclk_out	1	Output	Provides clock signal to the flash device
flash_ncs	1/3	Output	Provides the ncs signal to the flash device

The Intel FPGA Serial Flash Controller supports F_{MAX} at 25 MHz.

Note: When using the Intel FPGA Serial Flash Controller Core with an external EPCQ flash, the interface mapping for control signal is not required.

19.2. Registers

19.2.1. Register Memory Map

Each address offset in the table below represents 1 word of memory address space.

Table 162. Register Memory Map

Register	Offset	Width	Access	Description
FLASH_RD_STATUS	0x0	8	R	Perform read operation on flash device status register and store the read back data.
FLASH_RD_SID	0x1	8	R	Perform read operation to extract flash device silicon ID and store the read back data. Only support in EPCS16 and EPCS64 flash devices.
FLASH_RD_RDID	0x2	8	R	Perform read operation to extract flash device memory capacity and store the read back data.
FLASH_MEM_OP	0x3	24	W	To protect and erase memory
FLASH_ISR	0x4	2	RW	Interrupt status register
FLASH_IMR	0x5	2	RW	To mask of interrupt status register
FLASH_CHIP_SELECT	0x6	3	W	Chip select values: • B'000/b'001 -chip 1 • B'010 - chip 2 • B'100 - chip 3

19.2.2. Register Descriptions

19.2.2.1. FLASH_RD_STATUS

Table 163. FLASH_RD_STATUS

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								Read_status							

Table 164. FLASH_RD_STATUS Fields

Bit	Name	Description	Access	Default Value
31:8	Reserved	Reserved	-	0x0
7:0	Read_status	This 8 bits data contain the information from read status register operation. It keeps the information from the flash status register.	R	0x0

19.2.2.2. FLASH_RD_SID

Table 165. FLASH_RD_SID

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								Read_sid							

Table 166. FLASH_RD_SID Fields

Bit	Name	Description	Access	Default Value
31:8	Reserved	Reserved	-	0x0
7:0	Read_sid	This 8 bits data contain the information from read silicon ID operation.	R	0x0

19.2.2.3. FLASH_RD_RDID

Table 167. FLASH_RD_RDID

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								Read_rdid							

Table 168. FLASH_RD_RDID Fields

Bit	Name	Description	Access	Default Value
31:8	Reserved	Reserved	-	0x0
7:0	Read_rdid	This 8 bits data contain the information from read memory capacity operation. It keeps the information of the flash manufacturing ID.	R	0x0

19.2.2.4. FLASH_MEM_OP

Table 169. FLASH_MEM_OP

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								Sector value							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Sector value								Reserved						Memory protect/erase operation	

Table 170. FLASH_MEM_OP Fields

Bit	Name	Description	Access	Default Value
31:24	Reserved	Reserved	-	0x0
23:8	Sector value	Set the sector value of the flash device so that a particular memory sector can be erasing or protecting from erase or written. Please refer to the "Valid Sector Combination for Sector Protect and Sector Erase Command" section for more detail.	W	0x0
7:2	Reserved	Reserved	-	0x0
1:0	Memory protect/erase operation	• **2'b11 – Sector protect:** Active-high port that executes the sector protect operation. If asserted, the IP takes the value of FLASH_MEM_OP[23:8] and writes to the FLASH status register. The status register contains the block protection bits that represent the memory sector to be protected from write or erase. • **2'b10 – Sector erase:** Active-high port that executes the sector erase operation. If asserted, the IP starts erasing the memory sector on the flash device based on FLASH_MEM _OP[23:8] value. • **2'b01 – Bulk erase** Active-high port that executes the bulk erase operation. If asserted, the IP performs a full-erase operation that sets all memory bits of the flash device to '1', which includes the general purpose memory of the flash device. (Bulk erase is not supported in stack-die such as EPCQ512-L and EPCQ1024-L) • **2'b00 – N/A**	W	0x0

19.2.2.4.1. Valid Sector Combination for Sector Protect and Sector Erase Command

Sector Protect

For the sector protect command, you are allowed to perform the operation on more than one sector by giving the valid sector combination value to `FLASH_MEM_OP[23:8]` .

There are only 5 bits needed to provide the sector combination value. Bit 13 to bit 23 are reserved and should be set to zero.

Table 171. FLASH_MEM_OP bits for Sector Value

23	...	...		13	12	11	10	9	8
Reserved					TB	BP3	BP2	BP1	BP0

For more details about the sector combination values, refer to Quad-Serial Configuration (EPCQ) Devices Datasheet

Sector Erase

For the sector erase command, you are allowed to perform the operation on one sector at a time. Each sector contains of 65536 bytes of data, which is equivalent to 65536 address locations. You need to provide one sector value if you wish to erase to `FLASH_MEM_OP[23:8]` . For example, if you want to erase sector 127 in flash 256, you will need to assign `'b0000 0000 0111 1111` to `FLASH_MEM_OP[23:8]` .

Table 172. Number of sectors for different Flash Devices

	EPCQ16	EPCQ32	EPCQ64	EPCQ128	EPCQ256	EPCQ512	EPCQ1024
Valid sector range	0 to 31	0 to 63	0 to 127	0 to 255	0 to 511	0 to 1023	0 to 2047

19.2.2.5. FLASH_ISR

Table 173. FLASH_ISR

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved														Illegal write	Illegal erase

Table 174. FLASH_ISR Fields

Bit	Name	Description	Access	Default Value
31:2	Reserved	Reserved	-	0x0
1	Illegal write	Indicates that a write instruction is targeting a protected sector on the flash memory. This bit is set to indicate that the IP has cancelled a write instruction.	RW 1C	0x0
0	Illegal erase	Indicates that an erase instruction has been set to a protected sector on the flash memory. This bit is set to indicate that the IP has cancelled the erase instruction.	RW 1C	0x0

19.2.2.6. FLASH_IMR

Table 175. FLASH_IMR

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved														M_illegal_write	M_illegal_erase

Table 176. FLASH_IMR Fields

Bit	Name	Description	Access	Default Value
31:2	Reserved	Reserved	-	0x0
1	M_illegal_write	Mask bit for illegal write interrupt • 0: The corresponding interrupt is disabled • 1: The corresponding interrupt is enabled	RW	0x0
0	M_illegal_erase	Mask bit for illegal erase interrupt • 0: The corresponding interrupt is disabled • 1: The corresponding interrupt is enabled	RW	0x0

19.2.2.7. FLASH_CHIP_SELECT

Table 177. FLASH_CHIP_SELECT

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved													Chip_select bit 3	Chip_select bit 2	Chip_select bit 1

Table 178. FLASH_CHIP_SELECT Fields

Bit	Name	Description	Access	Default Value
31:3	Reserved	Reserved	-	0x0
2	Chip_select bit 3	In order to select flash chip 3, issue 1 to this bit while the rest of the bit to 0.	W	0x0
1	Chip_select bit 2	In order to select flash chip 2, issue 1 to this bit while the rest of the bit to 0.	W	0x0
0	Chip_select bit 1	In order to select flash chip 1, issue 1 or 0 to this bit while the rest of the bit to 0.	W	0x0

19.3. Nios II Tools Support

Related Information

- Using Flash Devices
- Using Flash Devices

19.3.1. Booting Nios II from Flash

Booting the Nios II from an flash will use a flow similar to Compact Flash Interface (CFI). The boot copier used will be the same one used for CFI flash.

The boot copier will be located in flash. This has a potential performance impact on the bootcopying process, which can be mitigated by using a flash cache.

There are two main scenarios when booting from flash:

- Executing in place

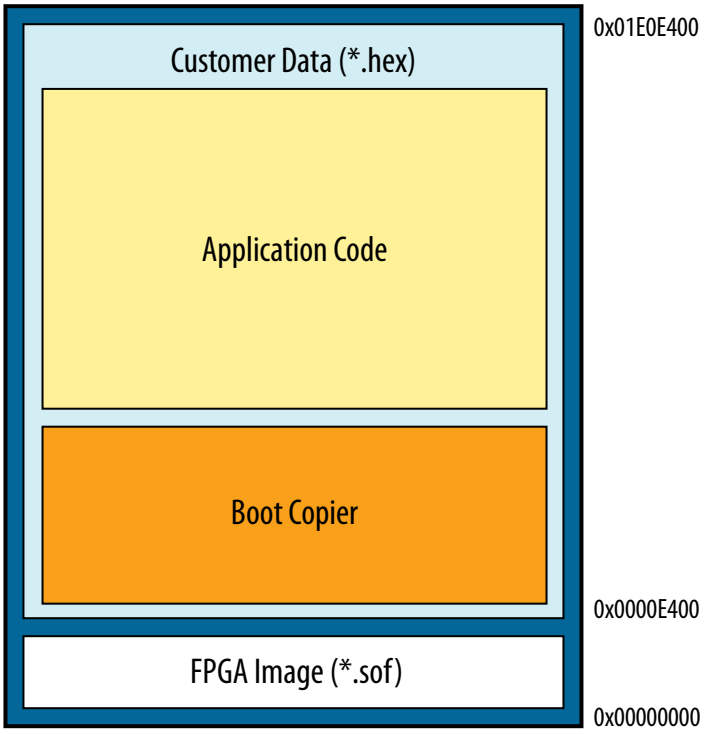
  In this scenario, boot copier will not be required. Nios II will directly execute customer code which located in flash.
- Boot copying the code to volatile memory

  In this scenario, boot copier is required. Nios II will run the boot copier code where the boot copier will copy customer code to volatile memory. This is normally used when customer concern about their code run time performance.

19.3.1.1. Flash Memory Map and Setting Nios II Reset Vector when Using a Boot Copier

The figure below shows what the flash memory map will look like when using a boot copier. This memory map assumes a FPGA image is stored at the start.

Figure 66. EPCQ Flash Layout When Using Boot Copier



At the start of the memory map is the FPGA image, followed by the boot copier, the application and then customer data. The size of the FPGA image is unknown and the exact size can only be known after the Intel Quartus Prime compile. However, the Nios II Reset Vector must be set in Platform Designer and must point to right after the FPGA image (i.e. the start of the boot copier).

The customer will have to determine an upper bound for the size of the FPGA image and will have to set the Nios II Reset Vector in Platform Designer to start after the FPGA image(s).

19.3.1.2. Boot Copier File

The boot copier that will be used is the CFI boot copier, also known as memcpy-based boot copier. We will provide the boot copier in one or more of the following formats: Intel HEX, Quartus HEX or SREC.

19.3.1.3. When Nios II SBT will Append a Boot Copier

The Nios II SBT tools know whether to append a boot copier based on the **.text** linker section location. If the **.text** linker section is located in a different memory than where the reset vector points, it indicates a code copy is required. At this scenario a boot copier is required. You can use the existing logic to generate a programming file with or without a boot copier depending on the scenario.

19.3.1.4. Creating HEX Programming File

The Nios II Software Build Tools (SBT) application Makefile "make mem_init_generate" target is responsible for generating memory initialization files. This includes generating programming files (SREC, HEX) used for flashing a flash memory and files for initializing memory (DAT, HEX) in simulation.

In boot scenario 1 (Executing in place), "make mem_init_generate" should generate a HEX file containing ELF loadable sections

In boot scenario 2 (Boot with copying the code to volatile memory), "make mem_init_generate" should generate a HEX file containing both the boot copier and ELF payload. "make mem_init_generate" is callable from SBT.

19.3.1.5. Programming the Flash

Programming the flash is done by using quartus_cpf to combine a compiled FPGA image (SOF) with an application image (HEX file generated by Nios II SBT). The result of this combination is a (POF) which can be programmed to the flash using the Intel Quartus Prime Programmer.

In the Intel Quartus Prime software, "Convert Programming File tool" (quartus_cpf) can be called by selecting `File >> Convert Programming Files`.

19.3.1.6. Custom Boot Copiers

Custom boot copiers can be used. "make mem_init_generate" calls conversion tools under the hood for creating programming files from compiled ELFs. These tools have a boot option to specify a custom boot copier. A user will need to call these underlying conversion tools to generate a programming file with a custom boot copier.

19.3.1.7. Executing in Place

Executing in place should not be any different than executing in place with an On-chip RAM. As long as both the Nios II reset and exception vectors point to the flash memory, execution will happen in place.

The Nios II board support package (BSP) settings are edited to enable `alt_load` function to copy the writable memory section into volatile memory and keep the read only section in the flash memory.

19.3.2. Nios II HAL Driver

The HAL API is available for this controller in the following software files:

- `altera_epcq_controller.h`
- `altera_epcq_controller.c`

These files implement the Serial Flash Controller core device driver for the HAL system library.

Nios II HAL supports a number of generic device model classes including one for device flashes. Developing against these generic classes gives a consistent interface for driver functions so that the HAL can access the driver functions uniformly.

Please refer to the *Flash Device Drivers* section in the *Developing Device Drivers for the Hardware Abstraction Layer* for more information.

Related Information

- Developing Device Drivers for the Hardware Abstraction Layer
- Nios II Configuration and Booting Solutions

19.4. Driver API

Table 179. alt_epcq_controller_lock

Prototype:	alt_epcq_controller_lock(alt_flash_dev *flash_info, alt_u32 sectors_to_lock)
Include:	<altera_epcq_controller.h>
Parameter:	• flash_info – pointer to general flash device structure • sector_to_lock- block protection bits in EPCQ
Return:	Return 0 if successful and otherwise return: • -EINVAL for Invalid argument • -ETIME for Time out and skipping the looping after 0.7sec • -ENOLCK for Sectors lock failed
Description:	Lock the range of memory sector which protected from write and erase.

Table 180. alt_epcq_controller_get_info

Prototype:	alt_epcq_controller_get_info(alt_flash_dev *fd, flash_region **info, int *number_of_regions)
Include:	<altera_epcq_controller.h>
Parameter:	• fd – pointer to general flash device structure • info- pointer to the flash region • number_of_regions- pointer to the number of regions
Return:	Return 0 if successful and otherwise return: • -EINVAL for Invalid argument • -EIO for possibly hardware problem
Description:	Pass the table of erase blocks to the user. The flash returns a single flash region that gives the number and size of sectors for the device used.

Table 181. alt_epcq_controller_erase_block

Prototype:	alt_epcq_controller_erase_block(alt_flash_dev *flash_info, int block_offset)
Include:	<altera_epcq_controller.h>
Parameter:	• flash_info – pointer to general flash device structure • block_offset- byte-addressed offset, from start of flash of the sector to be erased
Return:	Return 0 if successful and otherwise return: • -EINVAL for Invalid argument • -EIO for erase failed and sector might be protected
Description:	Erase a single flash sector.

Table 182. alt_epcq_controller_write_block

Prototype:	alt_epcq_controller_write_block(alt_flash_dev *flash_info, int block_offset, int data_offset, const void *data, int length)
Include:	<altera_epcq_controller.h>
Parameter:	• flash_info – pointer to general flash device structure • block_offset- byte-addressed offset, from start of flash of the sector to be written • data-offset – byte offset (unaligned access) of write into memory • data – data to be written • length – size of writing
Return:	Return 0 if successful and otherwise return: • -EINVAL for Invalid argument • -EIO for write failed and sector might be protected
Description:	Write one block/sector of data to the flash. The length of the write cannot spill into the adjacent sector,

Table 183. alt_epcq_controller_write

Prototype:	alt_epcq_controller_write (alt_flash_dev *flash_info, int offset, const void *src_addr, int length)
Include:	<altera_epcq_controller.h>
Parameter:	• flash_info – pointer to general flash device structure • offset- byte offset (unaligned access) of write to flash memory • src_addr – source buffer • length – size of writing
Return:	Return 0 if successful and otherwise return: • -EINVAL for Invalid argument • -EIO for write failed and sector might be protected
Description:	Program the data into the flash at the selected address. This function automatically erases a block as needed.

Table 184. alt_epcq_controller_read

Prototype:	alt_epcq_controller_read(alt_flash_dev *flash_info, int offset, void* dest_addr, int length)
Include:	<altera_epcq_controller.h>
Parameter:	• flash_info – pointer to general flash device structure • offset- byte offset read from flash memory • dest_addr – destination buffer • length – size of writing
Return:	Return 0 if successful and otherwise return: • -EINVAL for Invalid argument
Description:	Read data from flash at the selected address.

19.5. Intel FPGA Serial Flash Controller Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2021.03.29	21.1	Clarified support for Intel Stratix 10 and Intel Agilex devices.
2018.09.24	18.1	• Correct the list of supported *Configuration Device Types*. • Moved information about Controller II to a new chapter *Intel FPGA Serial Flash Controller II Core* • Added a new section: *Driver API*.
2018.05.07	18.0	• Added the F_{MAX} values for both Intel FPGA Serial Flash Controller and Controller II core. • Added the register descriptions for Intel FPGA Serial Flash Controller II core.

Date	Version	Changes
November 2017	2017.11.06	Added new configuration device types.
May 2017	2017.05.08	Added new section: *Intel FPGA Serial Flash Controller II core*
June 2015	2015.06.12	Initial release.

20. Intel FPGA Serial Flash Controller II Core

The Intel FPGA Serial Flash Controller II wraps around the Intel FPGA ASMI2 Parallel IP, and the logic that converts the ASMI2 Parallel conduit interface to Avalon interface.

The Intel FPGA Serial Flash Controller II supports F_{MAX} at 50 MHz, while the Intel FPGA Serial Flash Controller supports F_{MAX} at 25 MHz.

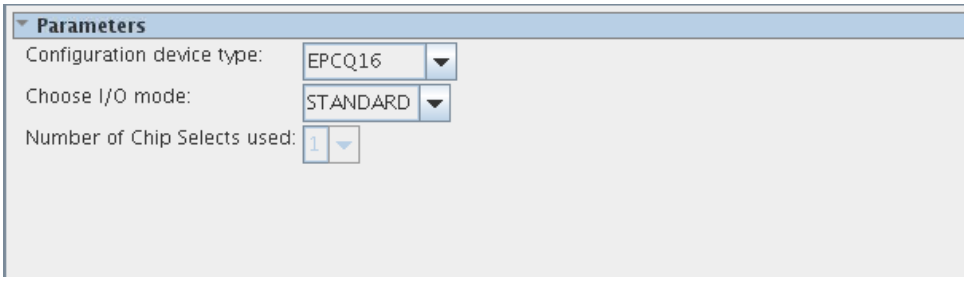
Note: The Intel FPGA Serial Flash Controller II core does not support Intel Stratix 10 and Intel Agilex devices.

Related Information

Generic Serial Flash Interface Intel FPGA IP Core User Guide
For your new (non Nios II) designs, Intel recommends you use the Generic Serial Flash Interface Intel FPGA IP core rather than this IP core.

20.1. Parameters

Figure 67. Platform Designer Parameters



20.1.1. Configuration Device Types

The following device types can be selected through the configuration device type drop down menu. Here you can specify the EPCQ.

- EPCQ16
- EPCQ32
- EPCQ64
- EPCQ128
- EPCQ256
- EPCQ512
- EPCQL256
- EPCQL512

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

- EPCQL1024
- EPCQ4A
- EPCQ16A
- EPCQ32A
- EPCQ64A
- EPCQ128A

While using EPCS, EPCQ, or EPCQL device for Intel FPGA Serial Flash Controller II, you must set MSEL pins for AS or ASx4 configuration scheme.

20.1.2. I/O Mode

From the parameters menu you can select either standard (AS configuration) or Quad (ASx4 configuration) I/O mode.

20.1.3. Chip Selects

For Intel Arria 10 devices, you can select up to three flash chips from the parameters menu.

20.1.4. Interface Signals

Table 185. Interface Signals

Signal	Width	Direction	Description
Clock			
clk	1	Input	50 MHz maximum input clock.
Reset			
reset_n	1	Input	Asynchronous reset used to reset Quad SPI Controller
Avalon-MM Agent Interface for CSR (avl_csr)			
avl_csr_addr	3	Input	Avalon-MM address bus. The address bus is in word addressing.
avl_csr_read	1	Input	Avalon-MM read control to csr
avl_csr_write	1	Input	Avalon-MM write control to csr
avl_csr_waitrequest	1	Output	Avalon-MM waitrequest control from csr
avl_csr_wrdata	32	Input	Avalon-MM write data bus to csr
avl_csr_rddata	32	Output	Avalon-MM read data bus from csr
avl_csr_rddata_valid	1	Output	Avalon-MM read data valid which indicates that csr read data is available

continued...

Signal	Width	Direction	Description
Avalon-MM Agent Interface for Memory Access (avl_ mem)			
avl_mem_addr	*	Input	Avalon-MM address bus. The address bus is in word addressing. The width of the address will depends on the flash memory density minus 2. If you are using Intel Arria 10, then the MSB bits will be used for chip select information. User is allowed to select the number of chip select needed in the GUI. If user selects 1 chip select, there will be no extra bit added to avl_mem_addr. If user select 2 chip selects, there will be one extra bit added to avl_mem_addr. Chip 1 – b'0 Chip 2 – b'1 If user select 3 chip selects, there will be two extra bit added to avl_mem_addr. Chip 1 – b'00 Chip 2 – b'01 Chip 3 – b'10
avl_mem_read	1	Input	Avalon-MM read control to memory
avl_mem_write	1	Input	Avalon-MM write control to memory
avl_mem_wrdata	32	Input	Avalon-MM write data bus to memory
avl_mem_byteenble	4	Input	Avalon-MM write data enable bit to memory. During bursting mode, byteenable bus bit will be all high always, 4'b1111.
avl_mem_burstcount	7	Input	Avalon-MM burst count for memory. Value range from 1 to 64
avl_mem_waitrequest	1	Output	Avalon-MM waitrequest control from memory
avl_mem_rddata	32	Output	Avalon-MM read data bus from memory
avl_mem_rddata_valid	1	Output	Avalon-MM read data valid which indicates that memory read data is available
Conduit Interface			
flash_dataout	4	Input/Output	Input/output port to feed data from flash device
flash_dclk_out	1	Output	Provides clock signal to the flash device
flash_ncs	1/3	Output	Provides the ncs signal to the flash device

20.2. Registers

20.2.1. Register Memory Map

Each address offset in the table below represents 1 word of memory address space.

Table 186. Register Memory Map

Register	Offset	Width	Access	Description
FLASH_RD_STATUS	0x0	8	R	Perform read operation on flash device status register and store the read back data.
FLASH_RD_RDID	0x2	8	R	Perform read operation to extract flash device memory capacity and store the read back data.
FLASH_MEM_OP	0x3	24	W	To protect, erase, and write enable memory.
FLASH_CHIP_SELECT	0x6	3	W	Chip select values: • b'000/b'001 - chip 1 • b'010 - chip 2 • b'100 - chip 3
EPCQ_FLAG_STATUS	0x7	8	RW	Perform write/read operation to clear/read flag status from the device.
DEVICE_ID_DATA_0	0x8	32	R	First word of device ID data
DEVICE_ID_DATA_1	0x9	32	R	Second word of device ID data
DEVICE_ID_DATA_2	0xA	32	R	Third word of device ID data
DEVICE_ID_DATA_3	0xB	32	R	Fourth word of device ID data
DEVICE_ID_DATA_4	0xC	32	R	Fifth word of device ID data

20.2.2. Register Descriptions

20.2.2.1. FLASH_RD_STATUS

Table 187. FLASH_RD_STATUS

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								Read_status							

Table 188. FLASH_RD_STATUS Fields

Bit	Name	Description	Access	Default Value
31:8	Reserved	Reserved	-	0x0
7:0	Read_status	This 8 bits data contain the information from read status register operation. It keeps the information from the flash status register.	R	0x0

20.2.2.2. FLASH_RD_RDID

Table 189. FLASH_RD_RDID

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								Read_rdid							

Table 190. FLASH_RD_RDID Fields

Bit	Name	Description	Access	Default Value
31:8	Reserved	Reserved	-	0x0
7:0	Read_rdid	This 8 bits data contain the information from read memory capacity operation. It keeps the information of the flash manufacturing ID.	R	0x0

20.2.2.3. FLASH_MEM_OP

Table 191. FLASH_MEM_OP

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								Sector value							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Sector value								Reserved					Memory protect/erase/ write enable operation		

Table 192. FLASH_MEM_OP Fields

Bit	Name	Description	Access	Default Value
31:18	Reserved	Reserved	-	0x0
23:8	Sector value	Set the sector value of the flash device so that a particular memory sector can be erasing or protecting from erase or written. Please refer to the *Valid Sector Value for Sector Protect and Sector Erase Command* section for more detail.	W	0x0
7:3	Reserved	Reserved	-	0x0
2:0	Memory protect/erase/ write enable operation	• **3'b011 – Sector protect:** Active-high port that executes the sector protect operation. If asserted, the IP takes the value of FLASH_MEM_OP[23:8] and writes to the FLASH status register. The status register contains the block protection bits that represent the memory sector to be protected from write or erase. • **3'b010 – Sector erase:** Active-high port that executes the sector erase operation. If asserted, the IP starts erasing the memory sector on the flash device based on FLASH_MEM _OP[23:8] value. • **3'b001 – Bulk erase** Active-high port that executes the bulk erase operation. If asserted, the IP performs a full-erase operation that sets all memory bits of the flash device to '1', which includes the general purpose memory of the flash device. (Bulk erase is not supported in stack-die such as EPCQ512-L and EPCQ1024-L) • **3'b100 – Write Enable Operation** All other options are N/A. For more information about how to perform the low level access operation, refer to the respective flash datasheet.	W	0x0

20.2.2.3.1. Valid Sector Combination for Sector Protect and Sector Erase Command

Sector Protect

For the sector protect command, you are allowed to perform the operation on more than one sector by giving the valid sector combination value to `FLASH_MEM_OP[23:8]` .

There are only 5 bits needed to provide the sector combination value. Bit 13 to bit 23 are reserved and should be set to zero.

Table 193. FLASH_MEM_OP bits for Sector Value

23	...	...		13	12	11	10	9	8
Reserved					TB	BP3	BP2	BP1	BP0

For more details about the sector combination values, refer to Quad-Serial Configuration (EPCQ) Devices Datasheet

Sector Erase

For the sector erase command, you are allowed to perform the operation on one sector at a time. Each sector contains of 65536 bytes of data, which is equivalent to 65536 address locations. You need to provide one sector value if you wish to erase to `FLASH_MEM_OP[23:8]` . For example, if you want to erase sector 127 in flash 256, you will need to assign `'b0000 0000 0111 1111` to `FLASH_MEM_OP[23:8]` .

Table 194. Number of sectors for different Flash Devices

	EPCQ16	EPCQ32	EPCQ64	EPCQ128	EPCQ256	EPCQ512	EPCQ1024
Valid sector range	0 to 31	0 to 63	0 to 127	0 to 255	0 to 511	0 to 1023	0 to 2047

20.2.2.4. FLASH_CHIP_SELECT

Table 195. FLASH_CHIP_SELECT

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved													Chip_ select bit 3	Chip_ select bit 2	Chip_ select bit 1

Table 196. FLASH_CHIP_SELECT Fields

Bit	Name	Description	Access	Default Value
31:3	Reserved	Reserved	-	0x0
2	Chip_select bit 3	In order to select flash chip 3, issue 1 to this bit while the rest of the bit to 0.	W	0x0
1	Chip_select bit 2	In order to select flash chip 2, issue 1 to this bit while the rest of the bit to 0.	W	0x0
0	Chip_select bit 1	In order to select flash chip 1, issue 1 or 0 to this bit while the rest of the bit to 0.	W	0x0

20.2.2.5. EPCQ_FLAG_STATUS

Table 197. EPCQ_FLAG_STATUS

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								flag_status							

Table 198. EPCQ_FLAG_STATUS Fields

Bit	Name	Description	Access	Default Value
31:8	Reserved	Reserved	-	0x0
7:0	flag_status[21]	This 8 bits data contain the information from read status register operation. It keeps the information from the flash status register. Clear the flag status register by writing value of 0 to the EPCQ_FLAG_STATUS register. Any value other than 0 is illegal.	RW	0x0

20.2.2.6. DEVICE_ID_DATA_*x*

Table 199. DEVICE_ID_DATA_*x*

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DEVICE_ID_DATA_*x*															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DEVICE_ID_DATA_*x*															

Table 200. DEVICE_ID_DATA_*x* Fields

Bit	Name	Description	Access	Default Value
31:0	DEVICE_ID_DATA_0	First word of device ID data	R	0x0
31:0	DEVICE_ID_DATA_1	Second word of device ID data	R	0x0
31:0	DEVICE_ID_DATA_2	Third word of device ID data	R	0x0
31:0	DEVICE_ID_DATA_3	Fourth word of device ID data	R	0x0
31:0	DEVICE_ID_DATA_4	Fifth word of device ID data	R	0x0

20.3. Nios II Tools Support

Related Information

- Using Flash Devices
- Using Flash Devices

20.3.1. Booting Nios II from Flash

Booting the Nios II from an flash will use a flow similar to Compact Flash Interface (CFI). The boot copier used will be the same one used for CFI flash.

The boot copier will be located in flash. This has a potential performance impact on the bootcopying process, which can be mitigated by using a flash cache.

[21] The Flag Status Register must be read every time a erase command is issued.

There are two main scenarios when booting from flash:

- Executing in place

  In this scenario, boot copier will not be required. Nios II will directly execute customer code which located in flash.

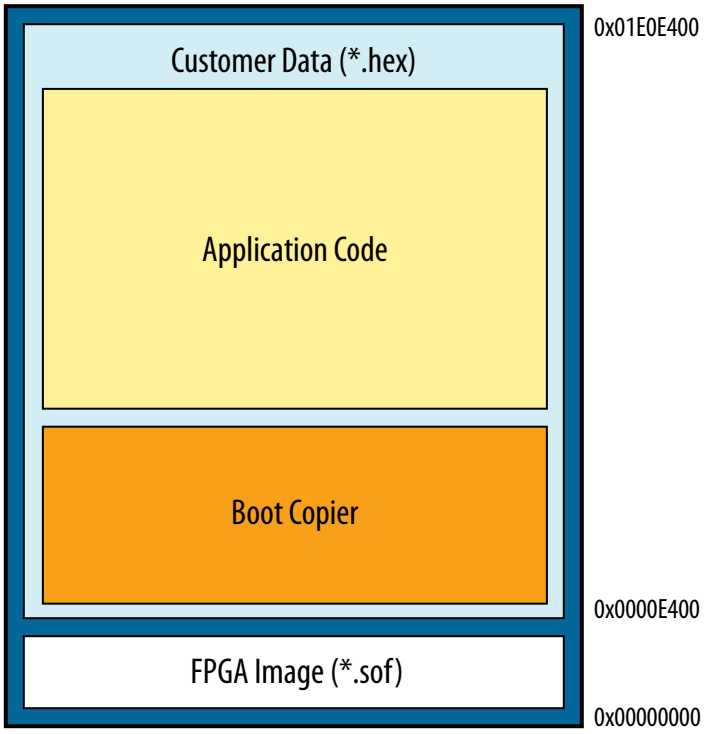
- Boot copying the code to volatile memory

  In this scenario, boot copier is required. Nios II will run the boot copier code where the boot copier will copy customer code to volatile memory. This is normally used when customer concern about their code run time performance.

20.3.1.1. Flash Memory Map and Setting Nios II Reset Vector when Using a Boot Copier

The figure below shows what the flash memory map will look like when using a boot copier. This memory map assumes a FPGA image is stored at the start.

Figure 68. EPCQ Flash Layout When Using Boot Copier



At the start of the memory map is the FPGA image, followed by the boot copier, the application and then customer data. The size of the FPGA image is unknown and the exact size can only be known after the Intel Quartus Prime compile. However, the Nios II Reset Vector must be set in Platform Designer and must point to right after the FPGA image (i.e. the start of the boot copier).

The customer will have to determine an upper bound for the size of the FPGA image and will have to set the Nios II Reset Vector in Platform Designer to start after the FPGA image(s).

20.3.1.2. Boot Copier File

The boot copier that will be used is the CFI boot copier, also known as memcpy-based boot copier. We will provide the boot copier in one or more of the following formats: Intel HEX, Quartus HEX or SREC.

20.3.1.3. When Nios II SBT will Append a Boot Copier

The Nios II SBT tools know whether to append a boot copier based on the **.text** linker section location. If the **.text** linker section is located in a different memory than where the reset vector points, it indicates a code copy is required. At this scenario a boot copier is required. You can use the existing logic to generate a programming file with or without a boot copier depending on the scenario.

20.3.1.4. Creating HEX Programming File

The Nios II Software Build Tools (SBT) application Makefile "make mem_init_generate" target is responsible for generating memory initialization files. This includes generating programming files (SREC, HEX) used for flashing a flash memory and files for initializing memory (DAT, HEX) in simulation.

In boot scenario 1 (Executing in place), "make mem_init_generate" should generate a HEX file containing ELF loadable sections

In boot scenario 2 (Boot with copying the code to volatile memory), "make mem_init_generate" should generate a HEX file containing both the boot copier and ELF payload. "make mem_init_generate" is callable from SBT.

20.3.1.5. Programming the Flash

Programming the flash is done by using quartus_cpf to combine a compiled FPGA image (SOF) with an application image (HEX file generated by Nios II SBT). The result of this combination is a (POF) which can be programmed to the flash using the Intel Quartus Prime Programmer.

In the Intel Quartus Prime software, "Convert Programming File tool" (quartus_cpf) can be called by selecting `File >> Convert Programming Files`.

20.3.1.6. Custom Boot Copiers

Custom boot copiers can be used. "make mem_init_generate" calls conversion tools under the hood for creating programming files from compiled ELFs. These tools have a boot option to specify a custom boot copier. A user will need to call these underlying conversion tools to generate a programming file with a custom boot copier.

20.3.1.7. Executing in Place

Executing in place should not be any different than executing in place with an On-chip RAM. As long as both the Nios II reset and exception vectors point to the flash memory, execution will happen in place.

The Nios II board support package (BSP) settings are edited to enable `alt_load` function to copy the writable memory section into volatile memory and keep the read only section in the flash memory.

20.3.2. Nios II HAL Driver

The HAL API is available for this controller in the following software files:

- `altera_epcq_controller2.h`
- `altera_epcq_controller2.c`

These files implement the Serial Flash Controller II core device driver for the HAL system library.

Nios II HAL supports a number of generic device model classes including one for device flashes. Developing against these generic classes gives a consistent interface for driver functions so that the HAL can access the driver functions uniformly.

Please refer to the *Flash Device Drivers* section in the *Developing Device Drivers for the Hardware Abstraction Layer* for more information.

20.4. Driver API

Table 201. alt_epcq_controller2_lock

Prototype:	alt_epcq_controller2_lock(alt_flash_dev *flash_info, alt_u32 sectors_to_lock)
Include:	<altera_epcq_controller2.h>
Parameter:	• flash_info – pointer to general flash device structure • sector_to_lock- block protection bits in EPCQ
Return:	Return 0 if successful and otherwise return: • -EINVAL for Invalid argument • -ETIME for Time out and skipping the looping after 0.7sec • -ENOLCK for Sectors lock failed
Description:	Lock the range of memory sector which protected from write and erase.

Table 202. alt_epcq_controller2_get_info

Prototype:	alt_epcq_controller2_get_info(alt_flash_dev *fd, flash_region **info, int *number_of_regions)
Include:	<altera_epcq_controller2.h>
Parameter:	• fd – pointer to general flash device structure • info- pointer to the flash region • number_of_regions- pointer to the number of regions
Return:	Return 0 if successful and otherwise return: • -EINVAL for Invalid argument • -EIO for possibly hardware problem
Description:	Pass the table of erase blocks to the user. The flash returns a single flash region that gives the number and size of sectors for the device used.

Table 203. alt_epcq_controller2_erase_block

Prototype:	alt_epcq_controller2_erase_block(alt_flash_dev *flash_info, int block_offset)
Include:	<altera_epcq_controller2.h>
	continued...

Parameter:	• flash_info – pointer to general flash device structure • block_offset- byte-addressed offset, from start of flash of the sector to be erased
Return:	Return 0 if successful and otherwise return: • -EINVAL for Invalid argument • -EIO for erase failed and sector might be protected
Description:	Erase a single flash sector.

Table 204. alt_epcq_controller2_write_block

Prototype:	alt_epcq_controller2_write_block(alt_flash_dev *flash_info, int block_offset, int data_offset, const void *data, int length)
Include:	<altera_epcq_controller2.h>
Parameter:	• flash_info – pointer to general flash device structure • block_offset- byte-addressed offset, from start of flash of the sector to be written • data-offset – byte offset (unaligned access) of write into memory • data – data to be written • length – size of writing
Return:	Return 0 if successful and otherwise return: • -EINVAL for Invalid argument • -EIO for write failed and sector might be protected
Description:	Write one block/sector of data to the flash. The length of the write cannot spill into the adjacent sector,

Table 205. alt_epcq_controller2_write

Prototype:	alt_epcq_controller2_write (alt_flash_dev *flash_info, int offset, const void *src_addr, int length)
Include:	<altera_epcq_controller2.h>
Parameter:	• flash_info – pointer to general flash device structure • offset- byte offset (unaligned access) of write to flash memory • src_addr – source buffer • length – size of writing
Return:	Return 0 if successful and otherwise return: • -EINVAL for Invalid argument • -EIO for write failed and sector might be protected
Description:	Program the data into the flash at the selected address. This function automatically erases a block as needed.

Table 206. alt_epcq_controller2_read

Prototype:	alt_epcq_controller2_read(alt_flash_dev *flash_info, int offset, void* dest_addr, int length)
Include:	<altera_epcq_controller2.h>
Parameter:	• flash_info – pointer to general flash device structure • offset- byte offset read from flash memory • dest_addr – destination buffer • length – size of writing
Return:	Return 0 if successful and otherwise return: • -EINVAL for Invalid argument
Description:	Read data from flash at the selected address.

20.5. Intel FPGA Serial Flash Controller II Core Revision History

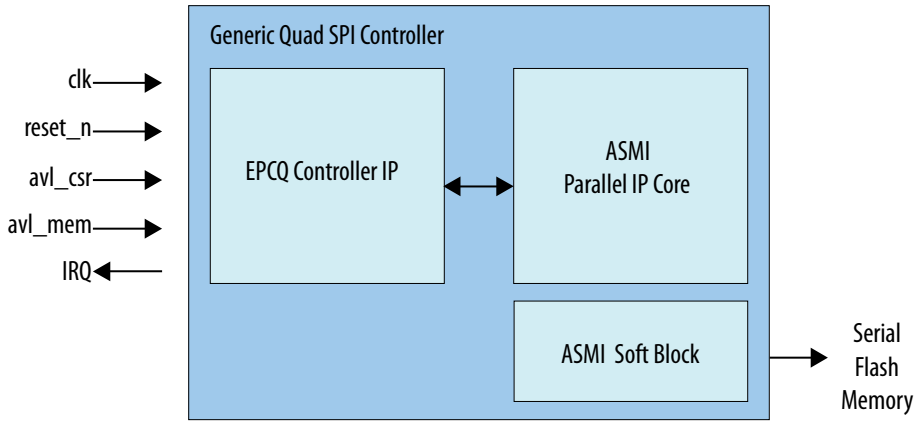
Document Version	Intel Quartus Prime Version	Changes
2021.03.29	21.1	Clarified support for Intel Stratix 10 and Intel Agilex devices.
2019.12.16	19.4	Removed the `irq` interrupt signal.
2019.04.01	19.1	Removed `FLASH_ISR` and `FLASH_IMR` registers.
2018.09.24	18.1	• Created this new chapter from the previous version's combined chapter: *Intel FPGA Serial Flash Controller and Controller II Core*. • Added a new section: *Driver API*.

21. Intel FPGA Generic QUAD SPI Controller Core

The Intel FPGA Generic QUAD SPI Controller Core wraps around the Intel FPGA ASMI PARALLEL IP, and a soft ASMI block. The flash interface is exported to the top wrapper.

Note: The Intel FPGA Generic QUAD SPI Controller core does not support Intel Stratix 10 and Intel Agilex devices.

Figure 69. Intel FPGA Generic QUAD SPI Controller Block Diagram

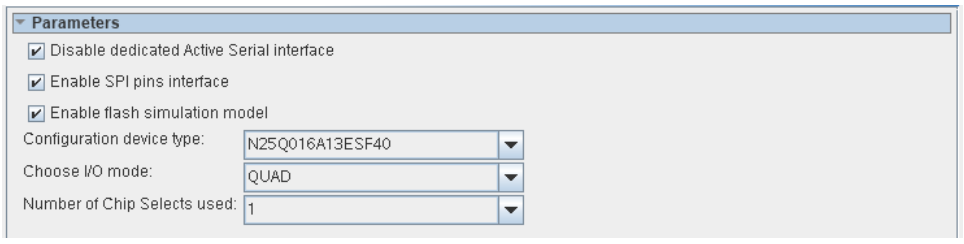


Related Information

Generic Serial Flash Interface Intel FPGA IP Core User Guide
For your new (non Nios II) designs, Intel recommends you use the Generic Serial Flash Interface Intel FPGA IP core rather than this IP core.

21.1. Parameters

Figure 70. Platform Designer Parameters



Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

21.1.1. Configuration Device Types

The following device types can be selected through the configuration device type drop down menu. Here you can specify the EPCQ or Micron flash type you want to use.

- EPCQ16
- EPCQ32
- EPCQ64
- EPCQ128
- EPCQ256
- EPCQ512
- EPCQL256
- EPCQL512
- EPCQL1024
- N25Q016A13ESF40
- N25Q032A13ESF40
- N25Q064A13ESF40
- N25Q128A13ESF40
- N25Q256A13ESF40
- N25Q256A13ESF40 (low voltage)
- MT25QL512ABA
- MT25QL512ABB
- N25Q512A11G1240 (low voltage)
- N25Q00AA11G1240 (low voltage)
- N25Q512A83GSF40F

While using EPCQ, or EPCQL device for Intel FPGA Generic QUAD SPI Controller, you must set MSEL pins for ASx4 configuration scheme.

21.1.2. I/O Mode

From the parameters menu you can select either standard or QUAD I/O mode.

21.1.3. Chip Selects

You can choose up to three flash chips from the parameter's menu.

Note: This feature is only for Intel Arria 10 and Intel Cyclone 10 GX devices.

21.1.4. Interface Signals

Table 207. Interface Signals

Signal	Width	Direction	Description
Clock			
clk	1	Input	up to 25MHz input clock.
Reset			
reset_n	1	Input	Asynchronous reset used to reset QUAD SPI controller
Avalon-MM Agent Interface for CSR (avl_csr)			
avl_csr_addr	3	Input	Avalon-MM address bus. The address bus is in word addressing.
avl_csr_read	1	Input	Avalon-MM read control to csr
avl_csr_write	1	Input	Avalon-MM write control to csr
avl_csr_waitrequest	1	Output	Avalon-MM waitrequest control from csr
avl_csr_wrdata	32	Input	Avalon-MM write data bus to csr
avl_csr_rddata	32	Output	Avalon-MM read data bus from csr
avl_csr_rddata_valid	1	Output	Avalon-MM read data valid which indicates that csr read data is available
Interrupt Signals			
irq	1	Output	Interrupt signal to determine if there is an illegal write or illegal erase
Avalon-MM Agent Interface for Memory Access (avl_ mem)			
avl_mem_addr	*	Input	Avalon-MM address bus. The address bus is in word addressing. The width of the address will depends on the flash memory density minus 2. If you are using Intel Arria 10, then the MSB bits will be used for chip select information. User is allowed to select the number of chip select needed in the GUI. If user selects 1 chip select, there will be no extra bit added to avl_mem_addr. If user select 2 chip selects, there will be one extra bit added to avl_mem_addr. Chip 1 – b'0 Chip 2 – b'1

continued...

Signal	Width	Direction	Description
			If user select 3 chip selects, there will be two extra bit added to avl_mem_addr. Chip 1 – b'00 Chip 2 – b'01 Chip 3 – b'10
avl_mem_read	1	Input	Avalon-MM read control to memory
avl_mem_write	1	Input	Avalon-MM write control to memory
avl_mem_wrdata	32	Input	Avalon-MM write data bus to memory
avl_mem_byteenble	4	Input	Avalon-MM write data enable bit to memory. During bursting mode, byteenable bus bit will be all high always, 4'b1111.
avl_mem_burstcount	7	Input	Avalon-MM burst count for memory. Value range from 1 to 64
avl_mem_waitrequest	1	Output	Avalon-MM waitrequest control from memory
avl_mem_rddata	32	Output	Avalon-MM read data bus from memory
avl_mem_rddata_valid	1	Output	Avalon-MM read data valid which indicates that memory read data is available
Conduit Interface			
flash_dataout	4	Input/Output	Input/output port to feed data from flash device
flash_dclk_out	1	Output	Provides clock signal to the flash device
flash_ncs	1/3	Output	Provides the ncs signal to the flash device

21.2. Registers

21.2.1. Register Memory Map

Each address offset in the table below represents 1 word of memory address space.

Table 208. Register Memory Map

Register	Offset	Width	Access	Description
FLASH_RD_STATUS	0x0	8	R	Perform read operation on flash device status register and store the read back data.
FLASH_RD_SID	0x1	8	R	Perform read operation to extract flash device silicon ID and store the read back data. Only support in EPCS16 and EPCS64 flash devices.

continued...

Register	Offset	Width	Access	Description
FLASH_RD_RDID	0x2	8	R	Perform read operation to extract flash device memory capacity and store the read back data.
FLASH_MEM_OP	0x3	24	W	To protect and erase memory
FLASH_ISR	0x4	2	RW	Interrupt status register
FLASH_IMR	0x5	2	RW	To mask of interrupt status register
FLASH_CHIP_SELECT	0x6	3	W	Chip select values: • B'000/b'001 -chip 1 • B'010 - chip 2 • B'100 - chip 3

21.2.2. Register Descriptions

21.2.2.1. FLASH_RD_STATUS

Table 209. FLASH_RD_STATUS

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								Read_status							

Table 210. FLASH_RD_STATUS Fields

Bit	Name	Description	Access	Default Value
31:8	Reserved	Reserved	-	0x0
7:0	Read_status	This 8 bits data contain the information from read status register operation. It keeps the information from the flash status register.	R	0x0

21.2.2.2. FLASH_RD_SID

Table 211. FLASH_RD_SID

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								Read_sid							

Table 212. FLASH_RD_SID Fields

Bit	Name	Description	Access	Default Value
31:8	Reserved	Reserved	-	0x0
7:0	Read_sid	This 8 bits data contain the information from read silicon ID operation.	R	0x0

21.2.2.3. FLASH_RD_RDID

Table 213. FLASH_RD_RDID

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								Read_rdid							

Table 214. FLASH_RD_RDID Fields

Bit	Name	Description	Access	Default Value
31:8	Reserved	Reserved	-	0x0
7:0	Read_rdid	This 8 bits data contain the information from read memory capacity operation. It keeps the information of the flash manufacturing ID.	R	0x0

21.2.2.4. FLASH_MEM_OP

Table 215. FLASH_MEM_OP

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								Sector value							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Sector value								Reserved						Memory protect/erase operation	

Table 216. FLASH_MEM_OP Fields

Bit	Name	Description	Access	Default Value
31:18	Reserved	Reserved	-	0x0
23:8	Sector value	Set the sector value of the flash device so that a particular memory sector can be erasing or protecting from erase or written. Please refer to the *Valid Sector Value for Sector Protect and Sector Erase Command* section for more detail.	W	0x0
7:2	Reserved	Reserved	-	0x0
1:0	Memory protect/erase operation	• **2'b11 – Sector protect:** Active-high port that executes the sector protect operation. If asserted, the IP takes the value of FLASH_MEM_OP[23:8] and writes to the FLASH status register. The status register contains the block protection bits that represent the memory sector to be protected from write or erase. • **2'b10 – Sector erase:** Active-high port that executes the sector erase operation. If asserted, the IP starts erasing the memory sector on the flash device based on FLASH_MEM _OP[23:8] value. • **2'b01 – Bulk erase** Active-high port that executes the bulk erase operation. If asserted, the IP performs a full-erase operation that sets all memory bits of the flash device to '1', which includes the general purpose memory of the flash device. (Bulk erase is not supported in stack-die such as EPCQ512-L and EPCQ1024-L) • **2'b00 – N/A**	W	0x0

21.2.2.4.1. Valid Sector Combination for Sector Protect and Sector Erase Command

Sector Protect

For the sector protect command, you are allowed to perform the operation on more than one sector by giving the valid sector combination value to `FLASH_MEM_OP[23:8]` .

There are only 5 bits needed to provide the sector combination value. Bit 13 to bit 23 are reserved and should be set to zero.

Table 217. FLASH_MEM_OP bits for Sector Value

23	...	...		13	12	11	10	9	8
Reserved					TB	BP3	BP2	BP1	BP0

For more details about the sector combination values, refer to Quad-Serial Configuration (EPCQ) Devices Datasheet

Sector Erase

For the sector erase command, you are allowed to perform the operation on one sector at a time. Each sector contains of 65536 bytes of data, which is equivalent to 65536 address locations. You need to provide one sector value if you wish to erase to FLASH_MEM_OP[23:8] . For example, if you want to erase sector 127 in flash 256, you will need to assign 'b0000 0000 0111 1111 to FLASH_MEM_OP[23:8] .

Table 218. Number of sectors for different Flash Devices

	EPCQ16	EPCQ32	EPCQ64	EPCQ128	EPCQ256	EPCQ512	EPCQ1024
Valid sector range	0 to 31	0 to 63	0 to 127	0 to 255	0 to 511	0 to 1023	0 to 2047

21.2.2.5. FLASH_ISR

Table 219. FLASH_ISR

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved														Illegal write	Illegal erase

Table 220. FLASH_ISR Fields

Bit	Name	Description	Access	Default Value
31:2	Reserved	Reserved	-	0x0
1	Illegal write	Indicates that a write instruction is targeting a protected sector on the flash memory. This bit is set to indicate that the IP has canceled a write instruction.	RW 1C	0x0
0	Illegal erase	Indicates that an erase instruction has been set to a protected sector on the flash memory. This bit is set to indicate that the IP has canceled the erase instruction.	RW 1C	0x0

21.2.2.6. FLASH_IMR

Table 221. FLASH_IMR

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved														M_illegal_write	M_illegal_erase

Table 222. FLASH_IMR Fields

Bit	Name	Description	Access	Default Value
31:2	Reserved	Reserved	-	0x0
1	M_illegal_write	Mask bit for illegal write interrupt • 0: The corresponding interrupt is disabled • 1: The corresponding interrupt is enabled	RW	0x0
0	M_illegal_erase	Mask bit for illegal erase interrupt • 0: The corresponding interrupt is disabled • 1: The corresponding interrupt is enabled	RW	0x0

21.2.2.7. FLASH_CHIP_SELECT

Table 223. FLASH_CHIP_SELECT

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved													Chip_select bit 3	Chip_select bit 2	Chip_select bit 1

Table 224. FLASH_CHIP_SELECT Fields

Bit	Name	Description	Access	Default Value
31:3	Reserved	Reserved	-	0x0
2	Chip_select bit 3	In order to select flash chip 3, issue 1 to this bit while the rest of the bit to 0.	W	0x0
1	Chip_select bit 2	In order to select flash chip 2, issue 1 to this bit while the rest of the bit to 0.	W	0x0
0	Chip_select bit 1	In order to select flash chip 1, issue 1 or 0 to this bit while the rest of the bit to 0.	W	0x0

21.3. Nios II Tools Support

Related Information

- Using Flash Devices
- Using Flash Devices
- Nios II Configuration and Booting Solutions
  For more information about booting Nios II from Flash.

21.3.1. Nios II HAL Driver

The HAL API is available for this controller in the following software files:

- `altera_generic_quad_spi_controller.h`
- `altera_generic_quad_spi_controller.c`

These files implement the Generic Quad SPI Controller core device driver for the HAL system library.

Nios II HAL supports a number of generic device model classes including one for device flashes. Developing against these generic classes gives a consistent interface for driver functions so that the HAL can access the driver functions uniformly.

Please refer to the *Flash Device Drivers* section in the *Developing Device Drivers for the Hardware Abstraction Layer* for more information.

Related Information

- Developing Device Drivers for the Hardware Abstraction Layer
- Using Flash Devices
  For more information about using the generic flash HAL driver to access flash.

21.4. Driver API

Table 225. alt_qspi_controller_lock

Prototype:	alt_qspi_controller_lock(alt_flash_dev *flash_info, alt_u32 sectors_to_lock)
Include:	<altera_qspi_controller.h>
Parameter:	• flash_info – pointer to general flash device structure • sector_to_lock- block protection bits in EPCQ
Return:	Return 0 if successful and otherwise return: • -EINVAL for Invalid argument • -ETIME for Time out and skipping the looping after 0.7sec • -ENOLCK for Sectors lock failed
Description:	Lock the range of memory sector which protected from write and erase.

Table 226. alt_qspi_controller_get_info

Prototype:	alt_qspi_controller_get_info(alt_flash_dev *fd, flash_region **info, int *number_of_regions)
Include:	<altera_qspi_controller.h>
Parameter:	• fd – pointer to general flash device structure • info- pointer to the flash region • number_of_regions- pointer to the number of regions
Return:	Return 0 if successful and otherwise return: • -EINVAL for Invalid argument • -EIO for possibly hardware problem
Description:	Pass the table of erase blocks to the user. The flash returns a single flash region that gives the number and size of sectors for the device used.

Table 227. alt_qspi_controller_erase_block

Prototype:	alt_qspi_controller_erase_block(alt_flash_dev *flash_info, int block_offset)
Include:	<altera_qspi_controller.h>
Parameter:	• flash_info – pointer to general flash device structure • block_offset- byte-addressed offset, from start of flash of the sector to be erased
Return:	Return 0 if successful and otherwise return:

continued...

	• -EINVAL for Invalid argument • -EIO for erase failed and sector might be protected
Description:	Erase a single flash sector.

Table 228. alt_qspi_controller_write_block

Prototype:	alt_qspi_controller_write_block(alt_flash_dev *flash_info, int block_offset, int data_offset, const void *data, int length)
Include:	<altera_qspi_controller.h>
Parameter:	• flash_info – pointer to general flash device structure • block_offset- byte-addressed offset, from start of flash of the sector to be written • data-offset – byte offset (unaligned access) of write into memory • data – data to be written • length – size of writing
Return:	Return 0 if successful and otherwise return: • -EINVAL for Invalid argument • -EIO for write failed and sector might be protected
Description:	Write one block/sector of data to the flash. The length of the write cannot spill into the adjacent sector,

Table 229. alt_qspi_controller_write

Prototype:	alt_qspi_controller_write (alt_flash_dev *flash_info, int offset, const void *src_addr, int length)
Include:	<altera_qspi_controller.h>
Parameter:	• flash_info – pointer to general flash device structure • offset- byte offset (unaligned access) of write to flash memory • src_addr – source buffer • length – size of writing
Return:	Return 0 if successful and otherwise return: • -EINVAL for Invalid argument • -EIO for write failed and sector might be protected
Description:	Program the data into the flash at the selected address. This function automatically erases a block as needed.

Table 230. alt_qspi_controller_read

Prototype:	alt_qspi_controller_read(alt_flash_dev *flash_info, int offset, void* dest_addr, int length)
Include:	<altera_qspi_controller.h>
Parameter:	• flash_info – pointer to general flash device structure • offset- byte offset read from flash memory • dest_addr – destination buffer • length – size of writing
Return:	Return 0 if successful and otherwise return: • -EINVAL for Invalid argument
Description:	Read data from flash at the selected address.

21.5. Intel FPGA Generic QUAD SPI Controller Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2021.03.29	21.1	Clarified support for Intel Stratix 10 and Intel Agilex devices.
2019.04.01	19.1	Added support for configuration device MT25QL512ABB.
2018.09.24	18.1	• Moved information about Controller II to a new chapter *Intel FPGA Generic QUAD SPI Controller II Core* • Added a new section: *Driver API*.
2018.05.07	18.0	• Corrected the list of *Configuration Device Types*. • Added the register descriptions for Intel FPGA Generic QUAD SPI Controller II core.

Date	Version	Changes
November 2017	2017.11.06	Added information about the write enable operation in *Table: Register Memory Map* for Intel FPGA Generic QUAD SPI Controller II core.
May 2017	2017.05.08	Added new section: *Intel FPGA Generic QUAD SPI Controller II core.*
June 2015	2015.06.12	Initial release.

22. Intel FPGA Generic QUAD SPI Controller II Core

The Generic QUAD SPI Controller II wraps around the Intel FPGA ASMI2 PARALLEL IP, and a soft ASMI block. The flash interface is exported to the top wrapper.

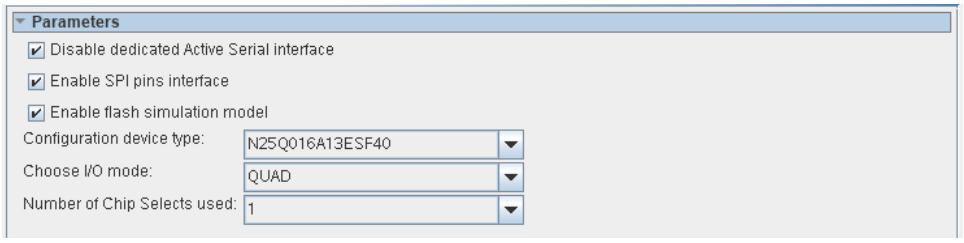
Note: The Intel FPGA Generic QUAD SPI Controller II core does not support Intel Agilex devices.

Related Information

- Generic Serial Flash Interface Intel FPGA IP Core User Guide
  For your new (non Nios II) designs, Intel recommends you use the Generic Serial Flash Interface Intel FPGA IP core rather than this IP core.
- AN 932: Flash Access Migration Guidelines from Control Block-Based Devices to SDM-Based Devices
  Provides guidelines on flash access when migrating from control block-based (Intel Arria 10 and V-Series) devices to SDM-based Intel Stratix 10 and Intel Agilex) devices.

22.1. Parameters

Figure 71. Platform Designer Parameters



22.1.1. Disable Dedicated Active Serial Interface

Enable this parameter to route ASMIBLOCK signals to top level of design when exporting flash pins to GPIO.

Note: This parameter is enabled by default. Starting Intel Quartus Prime software version 21.1, this parameter is disabled in Intel Stratix 10 devices.

22.1.2. Enable SPI Pins Interface

Enable this parameter to translate ASMIBLOCK signals to SPI pins interface.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

22.1.3. Enable Flash Simulation Model

Enable this parameter to use dedicated flash simulation model shipped with IP, otherwise you have to connect flash model manually.

Note: This parameter is only available in the Intel Quartus Prime Pro Edition.

22.1.4. Configuration Device Types

The following device types can be selected through the configuration device type drop down menu. Here you can specify the EPCQ or Micron flash type you want to use.

- EPCQ16[22]
- EPCQ32[22]
- EPCQ64[22]
- EPCQ128[22]
- EPCQ256[22]
- EPCQ512[22]
- EPCQL256[22]
- EPCQL512[22]
- EPCQL1024[22]
- N25Q016A13ESF40
- N25Q032A13ESF40
- N25Q064A13ESF40
- N25Q128A13ESF40
- N25Q256A13ESF40
- N25Q256A11E1240 (low voltage)
- MT25QL512ABA
- N25Q512A11G1240 (low voltage)
- N25Q00AA11G1240 (low voltage)
- N25Q512A83GSF40F
- MT25QL256
- MT25QL512
- MT25QU256
- MT25QU512
- MT25QU01G

While using EPCQ, or EPCQL device for Intel FPGA Generic QUAD SPI Controller II, you must set MSEL pins for ASx4 configuration scheme.

(22) Micron Compatible

22.1.5. I/O Mode

From the parameters menu you can select either standard or QUAD I/O mode.

22.1.6. Chip Selects

You can choose up to three flash chips from the parameter's menu.

Note: This feature is only for Intel Arria 10 and Intel Cyclone 10 GX devices.

22.2. Interface Signals

Table 231. Interface Signals

Signal	Width	Direction	Description
Clock			
clk	1	Input	up to 40MHz input clock.
Reset			
reset_n	1	Input	Asynchronous reset used to reset QUAD SPI controller
Avalon-MM Agent Interface for CSR (avl_csr)			
avl_csr_addr	3	Input	Avalon-MM address bus. The address bus is in word addressing.
avl_csr_read	1	Input	Avalon-MM read control to csr
avl_csr_write	1	Input	Avalon-MM write control to csr
avl_csr_waitrequest	1	Output	Avalon-MM waitrequest control from csr
avl_csr_wrdata	32	Input	Avalon-MM write data bus to csr
avl_csr_rddata	32	Output	Avalon-MM read data bus from csr
avl_csr_rddata_valid	1	Output	Avalon-MM read data valid which indicates that csr read data is available
Avalon-MM Agent Interface for Memory Access (avl_ mem)			
avl_mem_addr	*	Input	Avalon-MM address bus. The address bus is in word addressing. The width of the address will depends on the flash memory density minus 2. If you are using Intel Arria 10, then the MSB bits will be used for chip select information. User is allowed to select the number of chip select needed in the GUI. If user selects 1 chip select, there will be no extra bit added to avl_mem_addr. If user select 2 chip selects, there will be one extra bit added to avl_mem_addr. Chip 1 – b'0 Chip 2 – b'1 If user select 3 chip selects, there will be two extra bit added to avl_mem_addr. Chip 1 – b'00 Chip 2 – b'01 Chip 3 – b'10
avl_mem_read	1	Input	Avalon-MM read control to memory

continued...

Signal	Width	Direction	Description
avl_mem_write	1	Input	Avalon-MM write control to memory
avl_mem_wrdata	32	Input	Avalon-MM write data bus to memory
avl_mem_byteenable	4	Input	Avalon-MM write data enable bit to memory. During bursting mode, byteenable bus bit will be all high always, 4'b1111.
avl_mem_burstcount	7	Input	Avalon-MM burst count for memory. Value range from 1 to 64
avl_mem_waitrequest	1	Output	Avalon-MM waitrequest control from memory
avl_mem_rddata	32	Output	Avalon-MM read data bus from memory
avl_mem_rddata_valid	1	Output	Avalon-MM read data valid which indicates that memory read data is available
Conduit Interface			
flash_dataout	4	Input/Output	Input/output port to feed data from flash device
flash_dclk_out	1	Output	Provides clock signal to the flash device
flash_ncs	1/3	Output	Provides the ncs signal to the flash device
atom_ports_dclk(23)	1	Output	Provides clock signal to the flash device through ASMI block.
atom_ports_ncs(23)	1/3	Output	Provides the ncs signal to the flash device through ASMI block.
atom_ports_oe(23)	1	Output	Active-low signal to enable dclk and ncs pins to the flash through ASMI block.
atom_ports_dataout(23)	4	Output	Control signal from FPGA design to AS data pin for sending data into the flash through ASMI block.
atom_ports_dataoe(23)	4	Output	Controls AS data pin either as input or output: • b'0 – AS data pin as input • b'1 – AS data pin as output
atom_ports_datain(23)	4	Input	Signal from AS data pin to FPGA design through ASMI block.
qspi_pins_dclk(24)	1	Output	Provides clock signal to the flash device.
qspi_pins_ncs(24)	1/3	Output	Provides the ncs signal to the flash device.
qspi_pins_data(24)	4	Input/Output	Input/output port to feed data from flash device.

22.3. Registers

22.3.1. Register Memory Map

Each address offset in the table below represents 1 word of memory address space.

(23) Exported ASMI block signal.

(24) Exported SPI pin interface.

Table 232. Register Memory Map

Register	Offset	Width	Access	Description
FLASH_RD_STATUS	0x0	8	R	Perform read operation on flash device status register and store the read back data.
FLASH_RD_RDID	0x2	8	R	Perform read operation to extract flash device memory capacity and store the read back data.
FLASH_MEM_OP	0x3	24	W	To protect, erase, and write enable memory. To perform write enable operation, set this register with the value 3'b100.
FLASH_CHIP_SELECT	0x6	3	W	Chip select values: • B'000/b'001 -chip 1 • B'010 - chip 2 • B'100 - chip 3
EPCQ_FLAG_STATUS	0x7	8	RW	Perform write/read operation to clear/read flag status from the device.
DEVICE_ID_DATA_0	0x8	32	R	First word of device ID data
DEVICE_ID_DATA_1	0x9	32	R	Second word of device ID data
DEVICE_ID_DATA_2	0xA	32	R	Third word of device ID data
DEVICE_ID_DATA_3	0xB	32	R	Fourth word of device ID data
DEVICE_ID_DATA_4	0xC	32	R	Fifth word of device ID data

22.3.2. Register Descriptions

22.3.2.1. FLASH_RD_STATUS

Table 233. FLASH_RD_STATUS

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								Read_status							

Table 234. FLASH_RD_STATUS Fields

Bit	Name	Description	Access	Default Value
31:8	Reserved	Reserved	-	0x0
7:0	Read_status	This 8 bits data contain the information from read status register operation. It keeps the information from the flash status register.	R	0x0

22.3.2.2. FLASH_RD_RDID

Table 235. FLASH_RD_RDID

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								Read_rdid							

Table 236. FLASH_RD_RDID Fields

Bit	Name	Description	Access	Default Value
31:8	Reserved	Reserved	-	0x0
7:0	Read_rdid	This 8 bits data contain the information from read memory capacity operation. It keeps the information of the flash manufacturing ID.	R	0x0

22.3.2.3. FLASH_MEM_OP

Table 237. FLASH_MEM_OP

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								Sector value							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Sector value								Reserved					Memory protect/erase/ write enable operation		

Table 238. FLASH_MEM_OP Fields

Bit	Name	Description	Access	Default Value
31:18	Reserved	Reserved	-	0x0
23:8	Sector value	Set the sector value of the flash device so that a particular memory sector can be erasing or protecting from erase or written. Please refer to the *Valid Sector Value for Sector Protect and Sector Erase Command* section for more detail.	W	0x0
7:3	Reserved	Reserved	-	0x0
2:0	Memory protect/erase/ write enable operation	• **3'b011 – Sector protect:** Active-high port that executes the sector protect operation. If asserted, the IP takes the value of FLASH_MEM_OP[23:8] and writes to the FLASH status register. The status register contains the block protection bits that represent the memory sector to be protected from write or erase. • **3'b010 – Sector erase:** Active-high port that executes the sector erase operation. If asserted, the IP starts erasing the memory sector on the flash device based on FLASH_MEM _OP[23:8] value. • **3'b001 – Bulk erase** Active-high port that executes the bulk erase operation. If asserted, the IP performs a full-erase operation that sets all memory bits of the flash device to '1', which includes the general purpose memory of the flash device. (Bulk erase is not supported in stack-die such as EPCQ512-L and EPCQ1024-L) • **3'b100 – Write Enable Operation** All other options are N/A. For more information about how to perform the low level access operation, refer to the respective flash datasheet.	W	0x0

22.3.2.3.1. Valid Sector Combination for Sector Protect and Sector Erase Command

Sector Protect

For the sector protect command, you are allowed to perform the operation on more than one sector by giving the valid sector combination value to `FLASH_MEM_OP[23:8]` .

There are only 5 bits needed to provide the sector combination value. Bit 13 to bit 23 are reserved and should be set to zero.

Table 239. FLASH_MEM_OP bits for Sector Value

23	...	...		13	12	11	10	9	8
Reserved					TB	BP3	BP2	BP1	BP0

For more details about the sector combination values, refer to Quad-Serial Configuration (EPCQ) Devices Datasheet

Sector Erase

For the sector erase command, you are allowed to perform the operation on one sector at a time. Each sector contains of 65536 bytes of data, which is equivalent to 65536 address locations. You need to provide one sector value if you wish to erase to `FLASH_MEM_OP[23:8]` . For example, if you want to erase sector 127 in flash 256, you will need to assign `'b0000 0000 0111 1111` to `FLASH_MEM_OP[23:8]` .

Table 240. Number of sectors for different Flash Devices

	EPCQ16	EPCQ32	EPCQ64	EPCQ128	EPCQ256	EPCQ512	EPCQ1024
Valid sector range	0 to 31	0 to 63	0 to 127	0 to 255	0 to 511	0 to 1023	0 to 2047

22.3.2.4. FLASH_CHIP_SELECT

Table 241. FLASH_CHIP_SELECT

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved													Chip_ select bit 3	Chip_ select bit 2	Chip_ select bit 1

Table 242. FLASH_CHIP_SELECT Fields

Bit	Name	Description	Access	Default Value
31:3	Reserved	Reserved	-	0x0
2	Chip_select bit 3	In order to select flash chip 3, issue 1 to this bit while the rest of the bit to 0.	W	0x0
1	Chip_select bit 2	In order to select flash chip 2, issue 1 to this bit while the rest of the bit to 0.	W	0x0
0	Chip_select bit 1	In order to select flash chip 1, issue 1 or 0 to this bit while the rest of the bit to 0.	W	0x0

22.3.2.5. EPCQ_FLAG_STATUS

Table 243. EPCQ_FLAG_STATUS

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								flag_status							

Table 244. EPCQ_FLAG_STATUS Fields

Bit	Name	Description	Access	Default Value
31:8	Reserved	Reserved	-	0x0
7:0	flag_status[25]	This 8 bits data contain the information from read status register operation. It keeps the information from the flash status register. Clear the flag status register by writing value of 0 to the EPCQ_FLAG_STATUS register. Any value other than 0 is illegal.	RW	0x0

22.3.2.6. DEVICE_ID_DATA_*x*

Table 245. DEVICE_ID_DATA_*x*

Bit Fields															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DEVICE_ID_DATA_*x*															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DEVICE_ID_DATA_*x*															

Table 246. DEVICE_ID_DATA_*x* Fields

Bit	Name	Description	Access	Default Value
31:0	DEVICE_ID_DATA_0	First word of device ID data	R	0x0
31:0	DEVICE_ID_DATA_1	Second word of device ID data	R	0x0
31:0	DEVICE_ID_DATA_2	Third word of device ID data	R	0x0
31:0	DEVICE_ID_DATA_3	Fourth word of device ID data	R	0x0
31:0	DEVICE_ID_DATA_4	Fifth word of device ID data	R	0x0

22.4. Nios II Tools Support

Related Information

- Using Flash Devices
- Using Flash Devices
- Nios II Configuration and Booting Solutions
  For more information about booting Nios II from Flash.

22.4.1. Nios II HAL Driver

The HAL API is available for this controller in the following software files:

- `altera_generic_quad_spi_controller2.h`
- `altera_generic_quad_spi_controller2.c`

[25] The Flag Status Register must be read every time a erase command is issued.

These files implement the Generic Quad SPI Controller II core device driver for the HAL system library.

Nios II HAL supports a number of generic device model classes including one for device flashes. Developing against these generic classes gives a consistent interface for driver functions so that the HAL can access the driver functions uniformly.

Please refer to the *Flash Device Drivers* section in the *Developing Device Drivers for the Hardware Abstraction Layer* for more information.

22.5. Driver API

Table 247. alt_qspi_controller2_lock

Prototype:	alt_qspi_controller2_lock(alt_flash_dev *flash_info, alt_u32 sectors_to_lock)
Include:	<altera_qspi_controller2.h>
Parameter:	• flash_info – pointer to general flash device structure • sector_to_lock- block protection bits in EPCQ
Return:	Return 0 if successful and otherwise return: • -EINVAL for Invalid argument • -ETIME for Time out and skipping the looping after 0.7sec • -ENOLCK for Sectors lock failed
Description:	Lock the range of memory sector which protected from write and erase.

Table 248. alt_qspi_controller2_get_info

Prototype:	alt_qspi_controller2_get_info(alt_flash_dev *fd, flash_region **info, int *number_of_regions)
Include:	<altera_qspi_controller2.h>
Parameter:	• fd – pointer to general flash device structure • info- pointer to the flash region • number_of_regions- pointer to the number of regions
Return:	Return 0 if successful and otherwise return: • -EINVAL for Invalid argument • -EIO for possibly hardware problem
Description:	Pass the table of erase blocks to the user. The flash returns a single flash region that gives the number and size of sectors for the device used.

Table 249. alt_qspi_controller2_erase_block

Prototype:	alt_qspi_controller2_erase_block(alt_flash_dev *flash_info, int block_offset)
Include:	<altera_qspi_controller2.h>
Parameter:	• flash_info – pointer to general flash device structure • block_offset- byte-addressed offset, from start of flash of the sector to be erased
Return:	Return 0 if successful and otherwise return: • -EINVAL for Invalid argument • -EIO for erase failed and sector might be protected
Description:	Erase a single flash sector.

Table 250. alt_qspi_controller2_write_block

Prototype:	alt_qspi_controller2_write_block(alt_flash_dev *flash_info, int block_offset, int data_offset, const void *data, int length)
Include:	<altera_qspi_controller2.h>
Parameter:	• flash_info – pointer to general flash device structure • block_offset- byte-addressed offset, from start of flash of the sector to be written • data-offset – byte offset (unaligned access) of write into memory • data – data to be written • length – size of writing
Return:	Return 0 if successful and otherwise return: • -EINVAL for Invalid argument • -EIO for write failed and sector might be protected
Description:	Write one block/sector of data to the flash. The length of the write cannot spill into the adjacent sector,

Table 251. alt_qspi_controller2_write

Prototype:	alt_qspi_controller2_write (alt_flash_dev *flash_info, int offset, const void *src_addr, int length)
Include:	<altera_qspi_controller2.h>
Parameter:	• flash_info – pointer to general flash device structure • offset- byte offset (unaligned access) of write to flash memory • src_addr – source buffer • length – size of writing
Return:	Return 0 if successful and otherwise return: • -EINVAL for Invalid argument • -EIO for write failed and sector might be protected
Description:	Program the data into the flash at the selected address. This function automatically erases a block as needed.

Table 252. alt_qspi_controller2_read

Prototype:	alt_qspi_controller2_read(alt_flash_dev *flash_info, int offset, void* dest_addr, int length)
Include:	<altera_qspi_controller2.h>
Parameter:	• flash_info – pointer to general flash device structure • offset- byte offset read from flash memory • dest_addr – destination buffer • length – size of writing
Return:	Return 0 if successful and otherwise return: • -EINVAL for Invalid argument
Description:	Read data from flash at the selected address.

22.6. Intel FPGA Generic QUAD SPI Controller II Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2021.03.29	21.1	• Clarified support for Intel Agilex devices. • Added new parameters: — *Disable Dedicated Active Serial Interface* — *Enable SPI Pins Interface* — *Enable Flash Simulation Model* • Updated the list of supported *Configuration Device Types*. • Added new signals in section: *Interface Signals*.
2019.12.16	19.4	Removed the `irq` interrupt signal.
2019.04.01	19.1	• Added support for configuration device MT25QL512ABB. • Removed `FLASH_ISR` and `FLASH_IMR` registers.
2018.09.24	18.1	• Created this new chapter from the previous version's combined chapter: *Intel FPGA Generic QUAD SPI Controller and Controller II Core*. • Added a new section: *Driver API*.

23. Interval Timer Core

23.1. Core Overview

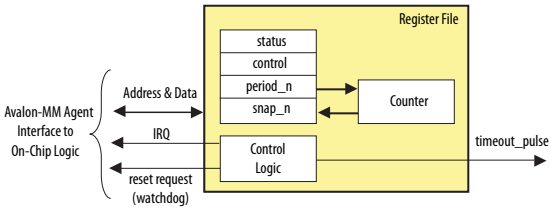
The Interval Timer core with Avalon interface is an interval timer for Avalon-based processor systems, such as a Nios II processor system. The core provides the following features:

- 32-bit and 64-bit counters.
- Controls to start, stop, and reset the timer.
- Two count modes: count down once and continuous count-down.
- Count-down period register.
- Option to enable or disable the interrupt request (IRQ) when timer reaches zero.
- Optional watchdog timer feature that resets the system if timer ever reaches zero.
- Optional periodic pulse generator feature that outputs a pulse when timer reaches zero.
- Compatible with 32-bit and 16-bit processors.

  Device drivers are provided in the HAL system library for the Nios II processor.

23.2. Functional Description

Figure 72. Interval Timer Core Block Diagram



Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

The interval timer core has two user-visible features:

- The Avalon Memory-Mapped (Avalon-MM) interface that provides access to six 16-bit registers
- An optional pulse output that can be used as a periodic pulse generator

  All registers are 16-bits wide, making the core compatible with both 16-bit and 32-bit processors. Certain registers only exist in hardware for a given configuration. For example, if the core is configured with a fixed period, the period registers do not exist in hardware.

  The following sequence describes the basic behavior of the interval timer core:

- An Avalon-MM host peripheral, such as a Nios II processor, writes the core's `control` register to perform the following tasks:
  - Start and stop the timer
  - Enable/disable the IRQ
  - Specify count-down once or continuous count-down mode
- A processor reads the `status` register for information about current timer activity.
- A processor can specify the timer period by writing a value to the period registers.
- An internal counter counts down to zero, and whenever it reaches zero, it is immediately reloaded from the period registers.
- A processor can read the current counter value by first writing to one of the snap registers to request a coherent snapshot of the counter, and then reading the snap registers for the full value.
- When the count reaches zero, one or more of the following events are triggered:
  - If IRQs are enabled, an IRQ is generated.
  - The optional pulse-generator output is asserted for one clock period.
  - The optional watchdog output resets the system.

23.2.1. Avalon-MM Agent Interface

The interval timer core implements a simple Avalon-MM agent interface to provide access to the register file. The Avalon-MM agent port uses the `resetrequest` signal to implement watchdog timer behavior. This signal is a non-maskable reset signal, and it drives the reset input of all Avalon-MM peripherals. When the `resetrequest` signal is asserted, it forces any processor connected to the system to reboot. For more information, refer to **Configuring the Timer as a Watchdog Timer**.

23.3. Configuration

This section describes the options available in the Platform Designer.

23.3.1. Timeout Period

The **Timeout Period** setting determines the initial value of the period registers. When the **Writeable period** option is on, a processor can change the value of the period by writing to the period registers. When the **Writeable period** option is off, the period is fixed and cannot be updated at runtime. See the **Hardware Options** section for information on register options.

The **Timeout Period** is an integer multiple of the **Timer Frequency**. The **Timer Frequency** is fixed at the frequency setting of the system clock associated with the timer. The **Timeout Period** setting can be specified in units of **µs** (microseconds), **ms** (milliseconds), **seconds** , or **clocks** (number of cycles of the system clock associated with the timer). The actual period depends on the frequency of the system clock associated with the timer. If the period is specified in **µs**, **ms**, or **seconds**, the true period will be the smallest number of clock cycles that is greater or equal to the specified **Timeout Period** value. For example, if the associated system clock has a frequency of 30 **ns**, and the specified **Timeout Period** value is 1 **µs**, the true timeout period will be 1.020 microseconds.

23.3.2. Counter Size

The **Counter Size** setting determines the timer's width, which can be set to either 32 or 64 bits. A 32-bit timer has two 16-bit period registers, whereas a 64-bit timer has four 16-bit period registers. This option applies to the snap registers as well.

23.3.3. Hardware Options

The following options affect the hardware structure of the interval timer core. As a convenience, the **Preset Configurations** list offers several pre-defined hardware configurations, such as:

- **Simple periodic interrupt**—This configuration is useful for systems that require only a periodic IRQ generator. The period is fixed and the timer cannot be stopped, but the IRQ can be disabled.
- **Full-featured**—This configuration is useful for embedded processor systems that require a timer with variable period that can be started and stopped under processor control.
- **Watchdog**—This configuration is useful for systems that require watchdog timer to reset the system in the event that the system has stopped responding. Refer to the **Configuring the Timer as a Watchdog Timer** section.

Register Options

Table 253. Register Options

Option	Description
Writeable period	When this option is enabled, a host peripheral can change the count-down period by writing to the period registers. When disabled, the count-down period is fixed at the specified **Timeout Period**, and the period registers do not exist in hardware.
Readable snapshot	When this option is enabled, a host peripheral can read a snapshot of the current count-down. When disabled, the status of the counter is detectable only via other indicators, such as the `status` register or the IRQ signal. In this case, the snap registers do not exist in hardware, and reading these registers produces an undefined value.
Start/Stop control bits	When this option is enabled, a host peripheral can start and stop the timer by writing the START and STOP bits in the `control` register. When disabled, the timer runs continuously. When the **System reset on timeout (watchdog)** option is enabled, the START bit is also present, regardless of the **Start/Stop control bits** option.

Output Signal Options

Table 254. Output Signal Options

Option	Description
Timeout pulse (1 clock wide)	When this option is on, the core outputs a signal `timeout_pulse`. This signal pulses high for one clock cycle whenever the timer reaches zero. When this option is off, the `timeout_pulse` signal does not exist.
System reset on timeout (watchdog)	When this option is on, the core's Avalon-MM agent port includes the `resetrequest` signal. This signal pulses high for one clock cycle whenever the timer reaches zero resulting in a system-wide reset. The internal timer is stopped at reset. Explicitly writing the START bit of the `control` register starts the timer. When this option is off, the `resetrequest` signal does not exist. Refer to the **Configuring the Timer as a Watchdog Timer** section.

23.3.4. Configuring the Timer as a Watchdog Timer

To configure the core for use as a watchdog, in the MegaWizard Interface select **Watchdog** in the **Preset Configurations** list, or choose the following settings:

- Set the **Timeout Period** to the desired "watchdog" period.
- Turn off **Writeable period**.
- Turn off **Readable snapshot**.
- Turn off **Start/Stop control bits**.
- Turn off **Timeout pulse**.
- Turn on **System reset on timeout (watchdog)**.

  A watchdog timer wakes up (comes out of reset) stopped. A processor later starts the timer by writing a 1 to the `control` register's START bit. Once started, the timer can never be stopped. If the internal counter ever reaches zero, the watchdog timer resets the system by generating a pulse on its `resetrequest` output. The `resetrequest` pulse will last for two cycles before the incoming reset signal deasserts the pulse. To prevent an indefinite `resetrequest` pulse, you are required to connect the `resetrequest` signal back to the reset input of the timer.

  To prevent the system from resetting, the processor must periodically reset the timer's count-down value by writing one of the period registers (the written value is ignored). If the processor fails to access the timer because, for example, software stopped executing normally, the watchdog timer resets the system and returns the system to a defined state.

23.4. Software Programming Model

The following sections describe the software programming model for the interval timer core, including the register map and software declarations to access the hardware. For Nios II processor users, Intel provides hardware abstraction layer (HAL) system library drivers that enable you to access the interval timer core using the HAL application programming interface (API) functions.

23.4.1. HAL System Library Support

The Intel-provided drivers integrate into the HAL system library for Nios II systems. When possible, HAL users should access the core via the HAL API, rather than accessing the core's registers directly.

Intel provides a driver for both the HAL timer device models: system clock timer, and timestamp timer.

System Clock Driver

When configured as the system clock, the interval timer core runs continuously in periodic mode, using the default period set. The system clock services are then run as a part of the interrupt service routine for this timer. The driver is interrupt-driven, and therefore must have its interrupt signal connected in the system hardware.

The Nios II integrated development environment (IDE) allows you to specify system library properties that determine which timer device will be used as the system clock timer.

Timestamp Driver

The interval timer core may be used as a timestamp device if it meets the following conditions:

- The timer has a writeable `period` register, as configured in Platform Designer.
- The timer is not selected as the system clock.

  The Nios II IDE allows you to specify system library properties that determine which timer device will be used as the timestamp timer.

  If the timer hardware is not configured with writeable `period` registers, calls to the `alt_timestamp_start()` API function will not reset the timestamp counter. All other HAL API calls will perform as expected.

  For more information about using the system clock and timestamp features that use these drivers, refer to the **Nios II Software Developer's Handbook**. The Nios II Embedded Design Suite (EDS) also provides several example designs that use the interval timer core.

Limitations

The HAL driver for the interval timer core does not support the watchdog reset feature of the core.

23.4.2. Software Files

The interval timer core is accompanied by the following software files. These files define the low-level interface to the hardware, and provide the HAL drivers. Application developers should not modify these files.

- **altera_avalon_timer_regs.h—**This file defines the core's register map, providing symbolic constants to access the low-level hardware.
- `altera_avalon_timer.h`, **altera_avalon_timer_sc.c, altera_avalon_timer_ts.c, altera_avalon_timer_vars.c—**These files implement the timer device drivers for the HAL system library.

23.4.3. Register Map

You do not need to access the interval timer core directly via its registers if using the standard features provided in the HAL system library for the Nios II processor. In general, the register map is only useful to programmers writing a device driver.

The Intel-provided HAL device driver accesses the device registers directly. If you are writing a device driver, and the HAL driver is active for the same device, your driver will conflict and fail to operate correctly.

The table below shows the register map for the 32-bit timer. The interval timer core uses native address alignment. For example, to access the `control` register value, use offset 0x4.

Table 255. Register Map—32-bit Timer

Offset	Name	R/W	Description of Bits						
			15	...	4	3	2	1	0
0	status	RW	(1)					RUN	TO
1	control	RW	(1)			STOP	START	CONT	ITO
2	periodl	RW	Timeout Period – 1 (bits [15:0])						
3	periodh	RW	Timeout Period – 1 (bits [31:16])						
4	snapl	RW	Counter Snapshot (bits [15:0])						
5	snaph	RW	Counter Snapshot (bits [31:16])						

Notes :
1. Reserved. Read values are undefined. Write zero.

For more information about native address alignment, refer to the System Interconnect Fabric for Memory-Mapped Interfaces.

Table 256. Register Map—64-bit Timer

Offset	Name	R/W	Description of Bits						
			15	...	4	3	2	1	0
0	status	RW	(1)					RUN	TO
1	control	RW	(1)			STOP	START	CONT	ITO
2	period_0	RW	Timeout Period – 1 (bits [15:0])						
3	period_1	RW	Timeout Period – 1 (bits [31:16])						
4	period_2	RW	Timeout Period – 1 (bits [47:32])						
5	period_3	RW	Timeout Period – 1 (bits [63:48])						
6	snap_0	RW	Counter Snapshot (bits [15:0])						
7	snap_1	RW	Counter Snapshot (bits [31:16])						
8	snap_2	RW	Counter Snapshot (bits [47:32])						
9	snap_3	RW	Counter Snapshot (bits [63:48])						

Notes :
1. Reserved. Read values are undefined. Write zero.

status Register

The `status` register has two defined bits.

Table 257. status Register Bits

Bit	Name	R/W/C	Description
0	TO	R/WC	The TO (timeout) bit is set to 1 when the internal counter reaches zero. Once set by a timeout event, the TO bit stays set until explicitly cleared by a host peripheral. Write 0 or 1 to the `status` register to clear the TO bit.
1	RUN	R	The RUN bit reads as 1 when the internal counter is running; otherwise this bit reads as 0. The RUN bit is not changed by a write operation to the `status` register.

control Register

The `control` register has four defined bits.

Table 258. control Register Bits

Bit	Name	R/W/C	Description
0	ITO	RW	If the ITO bit is 1, the interval timer core generates an IRQ when the `status` register's TO bit is 1. When the ITO bit is 0, the timer does not generate IRQs.
1	CONT	RW	The CONT (continuous) bit determines how the internal counter behaves when it reaches zero. If the CONT bit is 1, the counter runs continuously until it is stopped by the STOP bit. If CONT is 0, the counter stops after it reaches zero. When the counter reaches zero, it reloads with the value stored in the period registers, regardless of the CONT bit.
2	START (1)	W	Writing a 1 to the START bit starts the internal counter running (counting down). The START bit is an event bit that enables the counter when a write operation is performed. If the timer is stopped, writing a 1 to the START bit causes the timer to restart counting from the number currently stored in its counter. If the timer is already running, writing a 1 to START has no effect. Writing 0 to the START bit has no effect.
3	STOP (1)	W	Writing a 1 to the STOP bit stops the internal counter. The STOP bit is an event bit that causes the counter to stop when a write operation is performed. If the timer is already stopped, writing a 1 to STOP has no effect. Writing a 0 to the stop bit has no effect. If the timer hardware is configured with **Start/Stop control bits** off, writing the STOP bit has no effect.

Notes :
1. Writing 1 to both START and STOP bits simultaneously produces an undefined result.

period_n Registers

The `period_n` registers together store the timeout period value. The internal counter is loaded with the value stored in these registers whenever one of the following occurs:

- A write operation to one of the `period_n` register
- The internal counter reaches 0

  The timer's actual period is one cycle greater than the value stored in the `period_n` registers because the counter assumes the value zero for one clock cycle.

  Writing to one of the `period_n` registers stops the internal counter, except when the hardware is configured with **Start/Stop control bits** off. If **Start/Stop control bits** is off, writing either register does not stop the counter. When the hardware is configured with **Writeable period** disabled, writing to one of the `period_n` registers causes the counter to reset to the fixed **Timeout Period** specified at system generation time.

Note: A timeout period value of 0 is not a supported use case. Software configures timeout period values greater than 0.

snap_n Registers

A host peripheral may request a coherent snapshot of the current internal counter by performing a write operation (write-data ignored) to one of the `snap_n` registers. When a write occurs, the value of the counter is copied to `snap_n` registers. The snapshot occurs whether or not the counter is running. Requesting a snapshot does not change the internal counter's operation.

23.4.4. Interrupt Behavior

The interval timer core generates an IRQ whenever the internal counter reaches zero and the `ITO` bit of the `control` register is set to 1. Acknowledge the IRQ in one of two ways:

- Clear the `TO` bit of the `status` register
- Disable interrupts by clearing the `ITO` bit of the `control` register

  Failure to acknowledge the IRQ produces an undefined result.

23.5. Interval Time Core API

Table 259. alt_timestamp_start

Prototype	int alt_timestamp_start(void)
Include	<alt_timestamp.h>
Parameters	-
Returns	The return value is 0 upon success and -1 if timestamp device has not been registered
Description	Initialize the timestamp feature.

Table 260. alt_timestamp

Prototype	alt_timestamp_type alt_timestamp (void)
Include	<alt_timestamp.h>
Parameters	-
Returns	This function returns the current timestamp count. It returns -1 if the timer has run full period, or there is no timestamp available.
Description	Returns the current timestamp count.

Table 261. alt_timestamp_freq

Prototype	alt_u32 alt_timestamp_freq(void)
Include	<alt_timestamp.h>
Parameters	-
Returns	This function returns the number of timestamp ticks per second. This will be 0 if no timestamp device has been registered.
Description	Returns the number of timestamp ticks per second.

23.6. Interval Timer Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.09.24	18.1	Added a new section: *Interval Time Core API*.
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
June 2015	2015.06.12	Updated "status Register Bits" table.
July 2014	2014.07.24	Removed mention of SOPC Builder, updated to Platform Designer
December 2013	v13.1.0	Updated the reset pulse description in the **Configuring the Timer as a Watchdog Timer** section.
December 2010	v10.1.0	Removed the “Device Support”, “Instantiating the Core in SOPC Builder”, and “Referenced Documents” sections.
July 2010	v10.0.0	No change from previous release.
November 2009	v9.1.0	Revised descriptions of register fields and bits.
March 2009	v9.0.0	No change from previous release.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. Updated the core’s name to reflect the name used in SOPC Builder.
May 2008	v8.0.0	Added a new parameter and register map for the 64-bit timer.

24. Intel FPGA Avalon FIFO Memory Core

24.1. Core Overview

The Intel FPGA Avalon FIFO memory core buffers data and provides flow control in an Platform Designer system. The core can operate with a single clock or with separate clocks for the input and output ports, and it does not support burst read or write.

The input interface to the Intel FPGA Avalon FIFO memory core may be an Avalon Memory Mapped (Avalon-MM) write agent or an Avalon Streaming (Avalon-ST) sink. The output interface can be an Avalon-ST source or an Avalon-MM read agent. The data is delivered to the output interface in the same order that it was received at the input interface, regardless of the value of channel, packet, frame, or any other signals.

In single-clock mode, the Intel FPGA Avalon memory core includes an optional status interface that provides information about the fill level of the FIFO core. In dual-clock mode, separate, optional status interfaces can be included for the input and output interfaces. The status interface also includes registers to set and control interrupts.

Device drivers are provided in the HAL system library allowing software to access the core using ANSI C.

24.2. Functional Description

The Intel FPGA Avalon FIFO memory core has four configurations:

- Avalon Memory-Mapped Interface write agent to Avalon Memory-Mapped Interface read agent
- Avalon Streaming Interface sink to Avalon Streaming Interface source
- Avalon Memory-Mapped Interface write agent to Avalon Streaming Interface source
- Avalon Streaming Interface sink to Avalon Memory-Mapped Interface read agent

  In all configurations, the input and output interfaces can use the optional backpressure signals to prevent underflow and overflow conditions. For the Avalon Memory-Mapped Interface interface, backpressure is implemented using the `waitrequest` signal. For Avalon Streaming Interface interfaces, backpressure is implemented using the `ready` and `valid` signals. For the Intel FPGA Avalon FIFO memory core, the delay between the sink asserts `ready` and the source drives valid data is one cycle.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

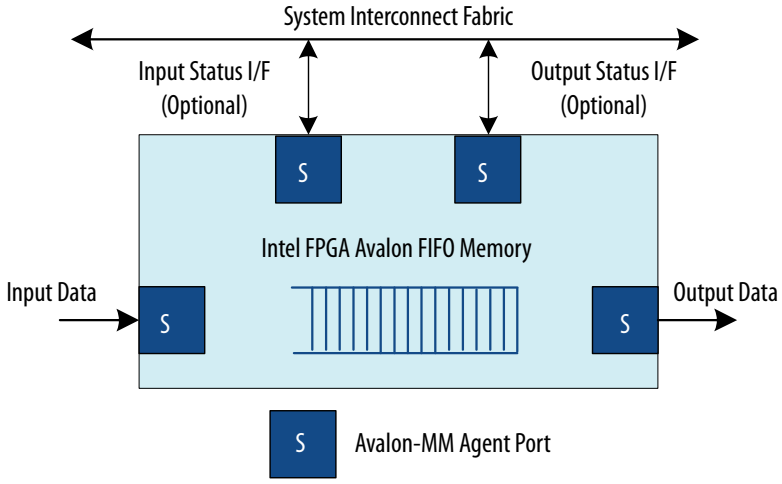
ISO 9001:2015 Registered

24.2.1. Avalon-MM Write Agent to Avalon-MM Read Agent

In this configuration, the input is a zero-address-width Avalon-MM write agent. An Avalon-MM write host pushes data into the FIFO core by writing to the input interface, and a read host (possibly the same host) pops data by reading from its output interface. The input and output data must be the same width.

If **Allow backpressure** is turned on, the `waitrequest` signal is asserted whenever the `data_in` host tries to write to a full FIFO buffer. `waitrequest` is only deasserted when there is enough space in the FIFO buffer for a new transaction to complete. `waitrequest` is asserted for read operations when there is no data to be read from the FIFO buffer, and is deasserted when the FIFO buffer has data. You must ensure that the FIFO is not empty using the `Empty` bit field of the status register before performing a read operation on FIFO output interface.

Figure 73. FIFO with Avalon-MM Input and Output Interfaces

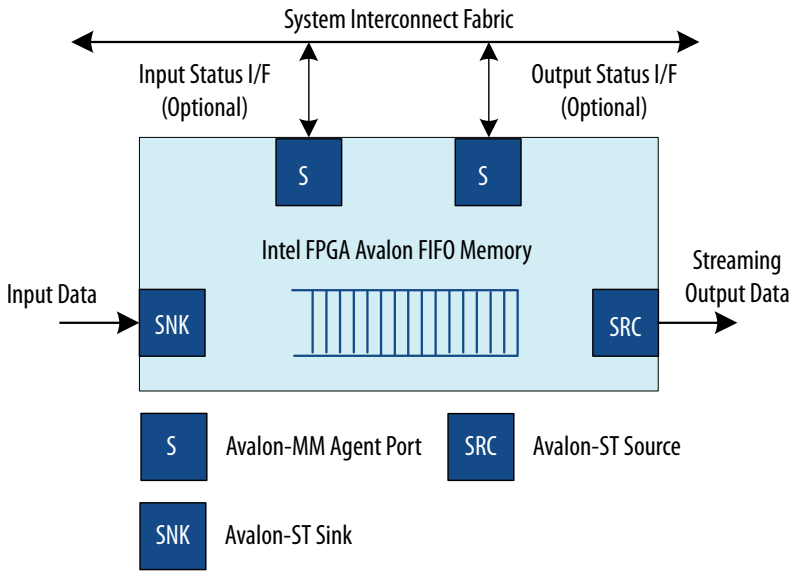


24.2.2. Avalon-ST Sink to Avalon-ST Source

This configuration has streaming input and output interfaces as illustrated in the figure below. You can parameterize most aspects of the Avalon-ST interfaces including the **bits per symbol**, **symbols per beat**, and the width of `error` and `channel` signals. The input and output interfaces must be the same width. If **Allow backpressure** is turned on, both interfaces use the `ready` and `valid` signals to indicate when space is available in the FIFO core and when valid data is available.

For more information about the Avalon-ST interface protocol, refer to the Avalon Interface Specifications.

Figure 74. FIFO with Avalon-ST Input and Output Interfaces



24.2.3. Avalon-MM Write Agent to Avalon-ST Source

In this configuration, the input is an Avalon-MM write agent with a width of 32 bits as shown in the **FIFO with Avalon-MM Input Interface and Avalon-ST Output Interface** figure below. The Avalon-ST output (source) data width must also be 32 bits. You can configure output interface parameters, including: **bits per symbol**, **symbols per beat**, and the width of the `channel` and `error` signals. The FIFO core performs the endian conversion to conform to the output interface protocol.

The signals that comprise the output interface are mapped into bits in the Avalon address space. If **Allow backpressure** is turned on, the input interface asserts `waitrequest` to indicate that the FIFO core does not have enough space for the transaction to complete.

Figure 75. FIFO with Avalon-MM Input Interface and Avalon-ST Output Interface

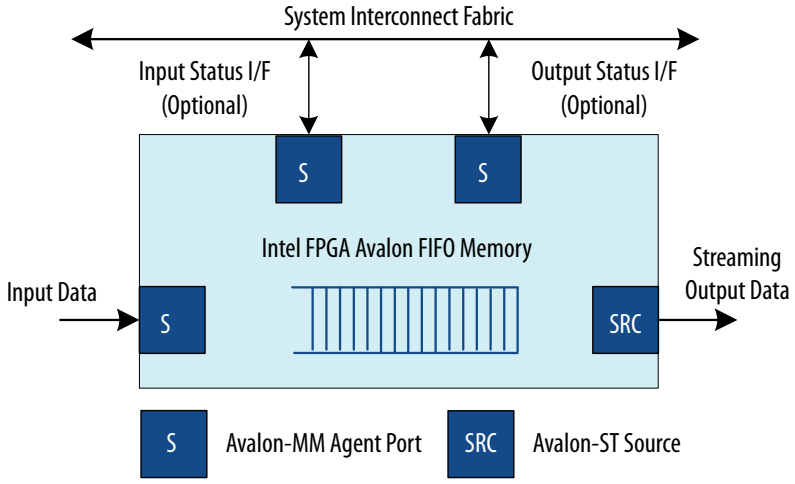


Table 262. Bit Field

Offset	31	24	23	19	18 16	15 13	12	8	7	4	3	2	1	0
base + 0	Symbol 3		Symbol 2			Symbol 1			Symbol 0					
base + 1	reserved		reserved		error	reserved	channel		reserved		empty		EOP	SOP

Table 263. Memory Map

Offset	Bits	Field	Description
0	31:0	SYMBOL_0, SYMBOL_1, SYMBOL_2 .. SYMBOL_n	Packet data. The value of the **Symbols per beat** parameter specifies the number of fields in this register; **Bits per symbol** specifies the width of each field.
1	0	SOP	The value of the `startofpacket` signal.
	1	EOP	The value of the `endofpacket` signal.
	6:2	EMPTY	The value of the `empty` signal.
	7	—	Reserved.
	15:8	CHANNEL	The value of the `channel` signal. The number of bits occupied corresponds to the width of the signal. For example, if the width of the channel signal is 5, bits 8 to 12 are occupied and bits 13 to 15 are unused.
	23:16	ERROR	The value of the `error` signal. The number of bits occupied corresponds to the width of the signal. For example, if the width of the error signal is 3, bits 16 to 18 are occupied and bits 19 to 23 are unused.
	31:24	—	Reserved.

If **Enable packet data** is turned off, the Avalon-MM write host writes all data at address offset 0 repeatedly to push data into the FIFO core.

If **Enable packet data** is turned on, the Avalon-MM write host starts by writing the SOP, ERROR (optional), CHANNEL (optional), EOP, and EMPTY packet status information at address offset 1. Writing to address offset 1 does not push data into the FIFO core. The Avalon-MM host then writes packet data to address offset 0 repeatedly, pushing 8-bit symbols into the FIFO core. Whenever a valid write occurs at address offset 0, the data and its respective packet information is pushed into the FIFO core. Subsequent data is written at address offset 0 without the need to clear the SOP field. Rewriting to address offset 1 is not required each time if the subsequent data to be pushed into the FIFO core is not the end-of-packet data, as long as ERROR and CHANNEL do not change.

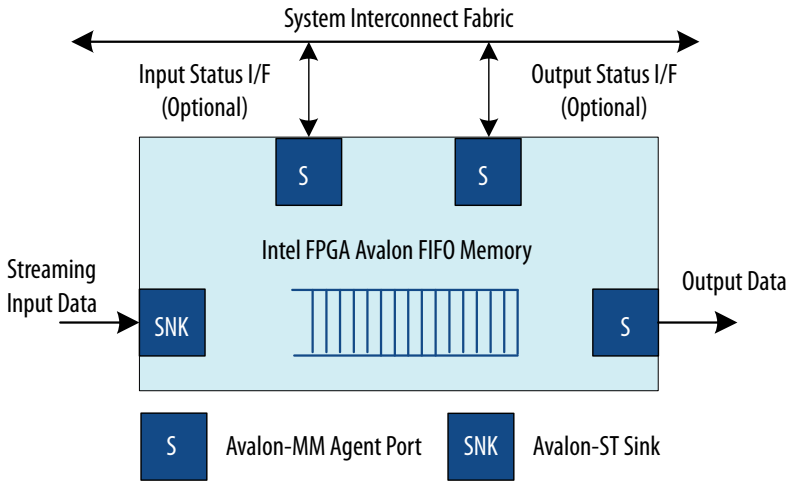
At the end of each packet, the Avalon-MM host writes to the address at offset 1 to set the EOP bit to 1, before writing the last symbol of the packet at offset 0. The write host uses the empty field to indicate the number of unused symbols at the end of the transfer. If the last packet data is not aligned with the **symbols per beat**, the EMPTY field indicates the number of empty symbols in the last packet data. For example, if the Avalon-ST interface has **symbols per beat** of 4, and the last packet only has 3 symbols, the empty field will be 1, indicating that one symbol (the least significant symbol in the memory map) is empty.

24.2.4. Avalon-ST Sink to Avalon-MM Read Agent

In this configuration seen in the figure below, the input is an Avalon-ST sink and the output is an Avalon-MM read agent with a width of 32 bits. The Avalon-ST input (sink) data width must also be 32 bits. You can configure input interface parameters, including: **bits per symbol**, **symbols per beat**, and the width of the `channel` and `error` signals. The FIFO core performs the endian conversion to conform to the output interface protocol.

An Avalon-MM host reads the data from the FIFO core. The signals are mapped into bits in the Avalon address space. If **Allow backpressure** is turned on, the input (sink) interface uses the `ready` and `valid` signals to indicate when space is available in the FIFO core and when valid data is available. For the output interface, `waitrequest` is asserted for read operations when there is no data to be read from the FIFO core. It is deasserted when the FIFO core has data to send. The memory map for this configuration is exactly the same as for the Avalon-MM to Avalon-ST FIFO core. See the for **Memory Map** table for more information.

Figure 76. FIFO with Avalon-ST Input and Avalon-MM Output



If **Enable packet data** is turned off, read data repeatedly at address offset 0 to pop the data from the FIFO core.

If **Enable packet data** is turned on, the Avalon-MM read host starts reading from address offset 0. If the read is valid, that is, the FIFO core is not empty, both data and packet status information are popped from the FIFO core. The packet status information is obtained by reading at address offset 1. Reading from address offset 1 does not pop data from the FIFO core. The ERROR, CHANNEL, SOP, EOP and EMPTY fields are available at address offset 1 to determine the status of the packet data read from address offset 0.

The EMPTY field indicates the number of empty symbols in the data field. For example, if the Avalon-ST interface has symbols-per-beat of 4, and the last packet data only has 1 symbol, the `empty` field is 3 to indicate that 3 symbols (the 3 least significant symbols in the memory map) are empty.

24.2.5. Status Interface

The FIFO core provides two optional status interfaces, one for the host writing to the input interface and a second for the read host reading from the output interface. For FIFO cores that operate in a single domain, a single status interface is sufficient to monitor the status of the FIFO core. In the dual clocking scheme, a second status interface using the output clock is necessary to accurately monitor the status of the FIFO core in both clock domains.

24.2.6. Clocking Modes

When single-clock mode is used, the FIFO core being used is SCFIFO. When dual-clock mode is chosen, the FIFO core being used is DCFIFO. In dual-clock mode, input data and write-side status interfaces use the write side clock domain; the output data and read-side status interfaces use the read-side clock domain.

24.3. Configuration

The following sections describe the available configuration options.

24.3.1. FIFO Settings

The following sections outline the settings that pertain to the FIFO core as a whole.

Depth

Depth indicates the depth of the FIFO buffer, in Avalon-ST bits or Avalon-MM words. The default depth is 16. When dual clock mode is used, the actual FIFO depth is equal to depth-3. This is due to clock crossing and to avoid FIFO overflow.

Clock Settings

The two options are **Single clock mode** and **Dual clock mode**. In **Single clock mode**, all interface ports use the same clock. In **Dual clock mode**, input data and input side status are on the input clock domain. Output data and output side status are on the output clock domain.

Status Port

The optional status ports are Avalon-MM agents. To include the optional input side status interface, turn on **Create status interface for input** on the Platform Designer MegaWizard. For FIFOs whose input and output ports operate in separate clock domains, you can include a second status interface by turning on **Create status interface for output**. Turning on **Enable IRQ for status ports** adds an interrupt signal to the status ports.

FIFO Implementation

This option determines if the FIFO core is built from registers or embedded memory blocks. The default is to construct the FIFO core from embedded memory blocks.

24.3.2. Interface Parameters

The following sections outline the options for the input and output interfaces.

Input

Available input interfaces are **Avalon Memory-Mapped Interface** write agent and **Avalon Streaming Interface** sink.

Output

Available output interfaces are **Avalon Memory-Mapped Interface** read agent and **Avalon Streaming Interface** source.

Allow Backpressure

When **Allow backpressure** is on, an Avalon Memory-Mapped Interface interface includes the `waitrequest` signal which is asserted to prevent a host from writing to a full FIFO buffer or reading from an empty FIFO buffer. An Avalon Streaming Interface interface includes the `ready` and `valid` signals to prevent underflow and overflow conditions.

Avalon Memory-Mapped Interface Port Settings

Valid **Data widths** are 8, 16, and 32 bits.

If Avalon Memory-Mapped Interface is selected for one interface and Avalon Streaming Interface for the other, the data width is fixed at 32 bits.

The Avalon Memory-Mapped Interface interface accesses data 4 bytes at a time. For data widths other than 32 bits, be careful of potential overflow and underflow conditions.

Avalon Streaming Interface Port Settings

The following parameters allow you to specify the size and error handling of the Avalon Streaming Interface port or ports:

- **Bits per symbol**
- **Symbols per beat**
- **Channel width**
- **Error width**

  If the symbol size is not a power of two, it is rounded up to the next power of two. For example, if the **bits per symbol** is 10, the symbol will be mapped to a 16-bit memory location. With 10-bit symbols, the maximum number of **symbols per beat** is two.

  Enable packet data provides an option for packet transmission.

24.3.3. Interface Signals

Table 264. Interface Signals

Signal	Width	Direction	Description
`wrclock`	1	Input	Clock signal for input interface.
`reset_n`	1	Input	Asynchronous reset signal that is associated with `wrclock`.

continued...

Signal	Width	Direction	Description
rdclock	1	Input	Clock signal for output interface. This signal is only available with the dual mode selection.
rdreset_n	1	Input	Asynchronous reset signal that is associated with rdclock. This signal is only available with the dual mode selection.
avalonmm_write_slave_write	1	Input	Write control signal. Enable when the input type is AVALONMM_WRITE.
avalonmm_write_slave_writedata	8-256	Input	Write data bus. Enable when the input type is AVALONMM_WRITE.
avalonst_sink_channel	8	Input	Channel bus. Enable when the input type is AVALONST_SINK.
avalonst_sink_data	32	Input	Data bus. Enable when the input type is AVALONST_SINK.
avalonst_sink_empty	1	Input	Empty signal. Enable when the input type is AVALONST_SINK.
avalonst_sink_endofpacket	1	Input	End of packet signal. Enable when the input type is AVALONST_SINK.
avalonst_sink_error	8	Input	Error signal. Enable when the input type is AVALONST_SINK.
avalonst_sink_startofpacket	1	Input	Start of packet signal. Enable when the input type is AVALONST_SINK.
avalonst_sink_valid	1	Input	Valid signal. Enable when the input type is AVALONST_SINK.
avalonmm_read_slave_read	1	Input	Read control signal. Enable when the input type is AVALONMM_READ.
avalonmm_read_slave_readdata	8-256	Output	Read data bus. Enable when the input type is AVALONMM_READ.
avalonst_source_channel	8	Output	Channel bus. Enable when the input type is AVALONST_SOURCE.
avalonst_source_data	32	Output	Data bus. Enable when the input type is AVALONST_SOURCE.
avalonst_source_empty	1	Output	Empty signal. Enable when the input type is AVALONST_SOURCE.
avalonst_source_endofpacket	1	Output	End of packet signal. Enable when the input type is AVALONST_SOURCE.
avalonst_source_error	8	Output	Error signal. Enable when the input type is AVALONST_SOURCE.
avalonst_source_startofpacket	1	Output	Start of packet signal. Enable when the input type is AVALONST_SOURCE.
avalonst_source_valid	1	Output	Valid signal. Enable when the input type is AVALONST_SOURCE.
wrclk_control_slave_address	3	Input	Address bus. Enable when the status interface for input is enabled.
wrclk_control_slave_read	1	Input	Read control signal. Enable when the status interface for input is enabled.

continued...

Signal	Width	Direction	Description
wrclk_control_slave_readdata	32	Output	Read data bus. Enable when the status interface for input is enabled.
wrclk_control_slave_write	1	Input	Write control signal. Enable when the status interface for input is enabled.
wrclk_control_slave_writedata	32	Input	Write data bus. Enable when the status interface for input is enabled.
rdclk_control_slave_address	3	Input	Address bus. Enable when the status interface for output is enabled.
rdclk_control_slave_read	1	Input	Read control signal. Enable when the status interface for output is enabled.
rdclk_control_slave_readdata	32	Output	Read data bus. Enable when the status interface for output is enabled.
rdclk_control_slave_write	1	Input	Write control signal. Enable when the status interface for output is enabled.
rdclk_control_slave_writedata	32	Input	Write data bus. Enable when the status interface for output is enabled.
wrclk_control_slave_irq	1	Output	Interrupt signal for input status register. Enable when IRQ and status interface for input are enabled.
rdclk_control_slave_irq	1	Output	Interrupt signal for output status register. Enable when IRQ and status interface for output are enabled.

24.4. Software Programming Model

The following sections describe the software programming model for the Intel FPGA Avalon FIFO memory core, including the register map and software declarations to access the hardware. For Nios II processor users, Intel provides HAL system library drivers that enable you to access the Intel FPGA Avalon FIFO memory core using its HAL API.

24.4.1. HAL System Library Support

The Intel-provided driver implements a HAL device driver that integrates into the HAL system library for Nios II systems. HAL users should access theFIFO memory via the familiar HAL API, rather than accessing the registers directly.

24.4.2. Software Files

Intel provides the following software files for the Intel FPGA Avalon FIFO memory core:

- `altera_avalon_fifo_regs.h`—This file defines the core's register map, providing symbolic constants to access the low-level hardware.
- `altera_avalon_fifo_util.h`—This file defines functions to access the on-chip FIFO memory core hardware. It provides utilities to initialize the FIFO, read and write status, enable flags and read events.
- `altera_avalon_fifo.h`—This file provides the public interface to the on-chip FIFO memory
- `altera_avalon_fifo_util.c`—This file implements the utilities listed in `altera_avalon_fifo_util.h`.

24.5. Programming with the Intel FPGA Avalon FIFO Memory

This section describes the low-level software constructs for manipulating the Intel FPGA Avalon FIFO memory core hardware. The table below lists all of the available functions.

Table 265. On-Chip FIFO Memory Functions

Function Name	Description
`altera_avalon_fifo_init()`	Initializes the FIFO.
`altera_avalon_fifo_read_status()`	Returns the integer value of the specified bit of the status register. To read all of the bits at once, use the `ALTERA_AVALON_FIFO_STATUS_ALL` mask.
`altera_avalon_fifo_read_ienable()`	Returns the value of the specified bit of the interrupt enable register. To read all of the bits at once, use the `ALTERA_AVALON_FIFO_EVENT_ALL` mask.
`altera_avalon_fifo_read_almostfull()`	Returns the value of the `almostfull` register.
`altera_avalon_fifo_read_almostempty()`	Returns the value of the `almostempty` register.
`altera_avalon_fifo_read_event()`	Returns the value of the specified bit of the event register. All of the event bits can be read at once by using the `ALTERA_AVALON_FIFO_STATUS_ALL` mask.
`altera_avalon_fifo_read_level()`	Returns the fill level of the FIFO.
`altera_avalon_fifo_clear_event()`	Clears the specified bits and the event register and performs error checking.
`altera_avalon_fifo_write_ienable()`	Writes the specified bits of the `interruptenable` register and performs error checking.
`altera_avalon_fifo_write_almostfull()`	Writes the specified value to the `almostfull` register and performs error checking.
`altera_avalon_fifo_write_almostempty()`	Writes the specified value to the `almostempty` register and performs error checking.
`altera_avalon_fifo_write_fifo()`	Writes the specified data to the `write_address`.
`altera_avalon_fifo_write_other_info()`	Writes the packet status information to the `write_address`. Only valid when the **Enable packet data** option is turned on.
`altera_avalon_fifo_read_fifo()`	Reads data from the specified `read_address`.
`altera_avalon_fifo_read__other_info()`	Reads the packet status information from the specified `read_address`. Only valid when the **Enable packet data** option is turned on.

24.5.1. Software Control

The table below provides the register map for the `status` register. The layout of `status` register for the input and output interfaces is identical.

Table 266. FIFO Status Register Memory Map

offset	31					24	23					16	15					8	7	6	5	4	3	2	1	0
base	`fill_level`																									
base + 1																					`i_status`				
base + 2																					`event`				
base + 3																					`interrupt enable`				
base + 4	`almostfull`																									
base + 5	`almostempty`																									

The table below outlines the use of the various fields of the

Table 267. FIFO Status Field Descriptions

Field	Type	Description
`fill_level`	RO	The instantaneous fill level of the FIFO, provided in units of symbols for a FIFO with an Avalon-ST FIFO and words for an Avalon-MM FIFO.
`i_status`	RO	A 6-bit register that shows the FIFO's instantaneous status. See **Status Bit Field Description** Table for the meaning of each bit field.
`event`	RW1C	A 6-bit register with exactly the same fields as `i_status`. When a bit in the `i_status` register is set, the same bit in the `event` register is set. The bit in the `event` register is only cleared when software writes a 1 to that bit.
`interruptenable`	RW	A 6-bit interrupt enable register with exactly the same fields as the `event` and `i_status` registers. When a bit in the `event` register transitions from a 0 to a 1, and the corresponding bit in `interruptenable` is set, the host Is interrupted.
`almostfull`	RW	A threshold level used for interrupts and status. Can be written by the Avalon-MM status host at any time. The default threshold value for DCFIFO is Depth-4. The default threshold value for SCFIFO is Depth-1. The valid range of the threshold value is from 1 to the default. 1 is used when attempting to write a value smaller than 1. The default is used when attempting to write a value larger than the default.
`almostempty`	RW	A threshold level used for interrupts and status. Can be written by the Avalon-MM status host at any time. The default threshold value for DCFIFO is 1. The default threshold value for SCFIFO is 1. The valid range of the threshold value is from 1 to the maximum allowable `almostfull` threshold. 1 is used when attempting to write a value smaller than 1. The maximum allowable is used when attempting to write a value larger than the maximum allowable.

`status` register.

Table 268. Status Bit Field Descriptions

Bit(s)	Name	Description
0	FULL	Has a value of 1 if the FIFO is currently full.
1	EMPTY	Has a value of 1 if the FIFO is currently empty.

continued...

Bit(s)	Name	Description
2	ALMOSTFULL	Has a value of 1 if the fill level of the FIFO is equal or greater than the `almostfull` value.
3	ALMOSTEMPTY	Has a value of 1 if the fill level of the FIFO is less or equal than the `almostempty` value.
4	OVERFLOW	Is set to 1 for 1 cycle every time the FIFO overflows. The FIFO overflows when an Avalon write host writes to a full FIFO. OVERFLOW is only valid when **Allow backpressure** is off.
5	UNDERFLOW	Is set to 1 for 1 cycle every time the FIFO underflows. The FIFO underflows when an Avalon read host reads from an empty FIFO. UNDERFLOW is only valid when **Allow backpressure** is off.

These fields are identical to those in the `status` register and are set at the same time; however, these fields are only cleared when software writes a one to clear (W1C). The `event` fields can be used to determine if a particular event has occurred.

Table 269. Event Bit Field Descriptions

Bit(s)	Name	Description
0	E_FULL	Has a value of 1 if the FIFO has been full and the bit has not been cleared by software.
1	E_EMPTY	Has a value of 1 if the FIFO has been empty and the bit has not been cleared by software.
2	E_ALMOSTFULL	Has a value of 1 if the fill level of the FIFO has been greater than the `almostfull` threshold value and the bit has not been cleared by software.
3	E_ALMOSTEMPTY	Has a value of 1 if the fill level of the FIFO has been less than the `almostempty` value and the bit has not been cleared by software.
4	E_OVERFLOW	Has a value of 1 if the FIFO has overflowed and the bit has not been cleared by software.
5	E_UNDERFLOW	Has a value of 1 if the FIFO has underflowed and the bit has not been cleared by software.

The table below provides a mask for the six STATUS fields. When a bit in the `event` register transitions from a zero to a one, and the corresponding bit in the `interruptenable` register is set, the host is interrupted.

Table 270. InterruptEnable Bit Field Descriptions

Bit(s)	Name	Description
0	IE_FULL	Enables an interrupt if the FIFO is currently full.
1	IE_EMPTY	Enables an interrupt if the FIFO is currently empty.
2	IE_ALMOSTFULL	Enables an interrupt if the fill level of the FIFO is greater than the value of the `almostfull` register.
3	IE_ALMOSTEMPTY	Enables an interrupt if the fill level of the FIFO is less than the value of the `almostempty` register.
4	IE_OVERFLOW	Enables an interrupt if the FIFO overflows. The FIFO overflows when an Avalon write host writes to a full FIFO.
5	IE_UNDERFLOW	Enables an interrupt if the FIFO underflows. The FIFO underflows when an Avalon read host reads from an empty FIFO.
6	ALL	Enables all 6 status conditions to interrupt.

Macros to access all of the registers are defined in **altera_avalon_fifo_regs.h.** For example, this file includes the following macros to access the `status` register.

```
#define ALTERA_AVALON_FIFO_LEVEL_REG       0
#define ALTERA_AVALON_FIFO_STATUS_REG      1
#define ALTERA_AVALON_FIFO_EVENT_REG       2
#define ALTERA_AVALON_FIFO_IENABLE_REG     3
#define ALTERA_AVALON_FIFO_ALMOSTFULL_REG  4
#define ALTERA_AVALON_FIFO_ALMOSTEMPTY_REG 5
```

For a complete list of predefined macros and utilities to access the on-chip FIFO hardware, see:

- `<install_dir>\quartus\sopc_builder\components\altera_avalon_fifo\HAL\inc\alatera_avalon_fifo.h`
- `<install_dir>\quartus\sopc_builder\components\altera_avalon_fifo\HAL\inc\alatera_avalon_fifo_util.h.`

24.5.2. Software Example

```
/*****************************************************************************/
//Includes
#include "altera_avalon_fifo_regs.h"
#include "altera_avalon_fifo_util.h"
#include "system.h"
#include "sys/alt_irq.h"
#include <stdio.h>
#include <stdlib.h>
#define ALMOST_EMPTY 2
#define ALMOST_FULL OUTPUT_FIFO_OUT_FIFO_DEPTH-5
volatile int input_fifo_wrclk_irq_event;
void print_status(alt_u32 control_base_address)
{
printf("---------------------------------------\n");
printf("LEVEL = %u\n", altera_avalon_fifo_read_level(control_base_address) );
printf("STATUS = %u\n", altera_avalon_fifo_read_status(control_base_address,
ALTERA_AVALON_FIFO_STATUS_ALL) );
printf("EVENT = %u\n", altera_avalon_fifo_read_event(control_base_address,
ALTERA_AVALON_FIFO_EVENT_ALL) );
printf("IENABLE = %u\n", altera_avalon_fifo_read_ienable(control_base_address,
ALTERA_AVALON_FIFO_IENABLE_ALL) );
printf("ALMOSTEMPTY = %u\n",
altera_avalon_fifo_read_almostempty(control_base_address) );
printf("ALMOSTFULL = %u\n\n",
altera_avalon_fifo_read_almostfull(control_base_address));
}
static void handle_input_fifo_wrclk_interrupts(void* context, alt_u32 id)
{
/* Cast context to input_fifo_wrclk_irq_event's type. It is important
* to declare this volatile to avoid unwanted compiler optimization.
*/
volatile int* input_fifo_wrclk_irq_event_ptr = (volatile int*) context;
/* Store the value in the FIFO's irq history register in *context. */
*input_fifo_wrclk_irq_event_ptr =
altera_avalon_fifo_read_event(INPUT_FIFO_IN_CSR_BASE,
ALTERA_AVALON_FIFO_EVENT_ALL);
printf("Interrupt Occurs for %#x\n", INPUT_FIFO_IN_CSR_BASE);
print_status(INPUT_FIFO_IN_CSR_BASE);
/* Reset the FIFO's IRQ History register. */
altera_avalon_fifo_clear_event(INPUT_FIFO_IN_CSR_BASE,
ALTERA_AVALON_FIFO_EVENT_ALL);
}
/* Initialize the fifo */
static int init_input_fifo_wrclk_control()
{
```

```
int return_code = ALTERA_AVALON_FIFO_OK;
/* Recast the IRQ History pointer to match the alt_irq_register() function
* prototype. */
void* input_fifo_wrclk_irq_event_ptr = (void*) &input_fifo_wrclk_irq_event;
/* Enable all interrupts. */
/* Clear event register, set enable all irq, set almostempty and
almostfull threshold */
return_code = altera_avalon_fifo_init(INPUT_FIFO_IN_CSR_BASE,
0, // Disabled interrupts
ALMOST_EMPTY,
ALMOST_FULL);
/* Register the interrupt handler. */
alt_irq_register( INPUT_FIFO_IN_CSR_IRQ,
input_fifo_wrclk_irq_event_ptr, handle_input_fifo_wrclk_interrupts );
return return_code;
}
```

24.6. Intel FPGA Avalon FIFO Memory API

This section describes the application programming interface (API) for the Intel FPGA Avalon FIFO memory core.

24.6.1. altera_avalon_fifo_init()

Prototype:	`int altera_avalon_fifo_init(alt_u32 address, alt_u32 ienable, alt_u32 emptymark, alt_u32 fullmark)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control agent `ienable`—the value to write to the `interruptenable` register `emptymark`—the value for the almost empty threshold level `fullmark`—the value for the almost full threshold level
Returns:	Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_EVENT_CLEAR_ERROR` for clear errors, `ALTERA_AVALON_FIFO_IENABLE_WRITE_ERROR` for interrupt enable write errors, `ALTERA_AVALON_FIFO_THRESHOLD_WRITE_ERROR` for errors writing the `almostfull` and `almostempty` registers.
Description:	Clears the `event` register, writes the `interruptenable` register, and sets the `almostfull` register and `almostempty` registers.

24.6.2. altera_avalon_fifo_read_status()

Prototype:	`int altera_avalon_fifo_read_status(alt_u32 address, alt_u32 mask)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control agent `mask`—masks the read value from the `status` register
Returns:	Returns the masked bits of the addressed register.
Description:	Gets the addressed register bits—the AND of the value of the addressed register and the mask.

24.6.3. altera_avalon_fifo_read_ienable()

Prototype:	`int altera_avalon_fifo_read_ienable(alt_u32 address, alt_u32 mask)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control agent `mask`—masks the read value from the `interruptenable` register
Returns:	Returns the logical AND of the `interruptenable` register and the mask.
Description:	Gets the logical AND of the `interruptenable` register and the mask.

24.6.4. altera_avalon_fifo_read_almostfull()

Prototype:	`int altera_avalon_fifo_read_almostfull(alt_u32 address)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control agent
Returns:	Returns the value of the `almostfull` register.
Description:	Gets the value of the `almostfull` register.

24.6.5. altera_avalon_fifo_read_almostempty()

Prototype:	`int altera_avalon_fifo_read_almostempty(alt_u32 address)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control agent
Returns:	Returns the value of the `almostempty` register.
Description:	Gets the value of the `almostempty` register.

24.6.6. altera_avalon_fifo_read_event()

Prototype:	`int altera_avalon_fifo_read_event(alt_u32 address, alt_u32 mask)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control agent

continued...

	`mask`—masks the read value from the `event` register
Returns:	Returns the logical AND of the `event` register and the mask.
Description:	Gets the logical AND of the `event` register and the mask. To read single bits of the event register use the single bit masks, for example: `ALTERA_AVALON_FIFO_FIFO_EVENT_F_MSK`. To read the entire `event` register use the full mask: `ALTERA_AVALON_FIFO_EVENT_ALL`.

24.6.7. altera_avalon_fifo_read_level()

Prototype:	`int altera_avalon_fifo_read_level(alt_u32 address)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control agent
Returns:	Returns the fill level of the FIFO.
Description:	Gets the fill level of the FIFO.

24.6.8. altera_avalon_fifo_clear_event()

Prototype:	`int altera_avalon_fifo_clear_event(alt_u32 address, alt_u32 mask)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control agent `mask`—the mask to use for bit-clearing (1 means clear this bit, 0 means do not clear)
Returns:	Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_EVENT_CLEAR_ERROR` if unsuccessful.
Description:	Clears the specified bits of the `event` register.

24.6.9. altera_avalon_fifo_write_ienable()

Prototype:	`int altera_avalon_fifo_write_ienable(alt_u32 address, alt_u32 mask)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control agent `mask`—the value to write to the `interruptenable` register. See **altera_avalon_fifo_regs.h** for individual interrupt bit masks.
Returns:	Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_IENABLE_WRITE_ERROR` if unsuccessful.
Description:	Writes the specified bits of the `interruptenable` register.

24.6.10. altera_avalon_fifo_write_almostfull()

Prototype:	`int altera_avalon_fifo_write_almostfull(alt_u32 address, alt_u32 data)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control agent `data`—the value for the almost full threshold level
Returns:	Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_THRESHOLD_WRITE_ERROR` if unsuccessful.
Description:	Writes `data` to the `almostfull` register.

24.6.11. altera_avalon_fifo_write_almostempty()

Prototype:	`int altera_avalon_fifo_write_almostempty(alt_u32 address, alt_u23 data)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`address`—the base address of the FIFO control agent `data`—the value for the almost empty threshold level
Returns:	Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_THRESHOLD_WRITE_ERROR` if unsuccessful.
Description:	Writes `data` to the `almostempty` register.

24.6.12. altera_avalon_write_fifo()

Prototype:	`int altera_avalon_write_fifo(alt_u32 write_address, alt_u32 ctrl_address, alt_u32 data)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`write_address`—the base address of the FIFO write agent `ctrl_address`—the base address of the FIFO control agent to read the status `data`—the value to write to address offset 0 for Avalon-MM to Avalon-ST transfers, the value to write to the single address available for Avalon-MM to Avalon-MM transfers. See the **Avalon Interface Specifications** section for the data ordering.
Returns:	Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_FULL` if unsuccessful.
Description:	Writes `data` to the specified address if the FIFO is not full.

24.6.13. altera_avalon_write_other_info()

Prototype:	`int altera_avalon_write_other_info(alt_u32 write_address, alt_u32 ctrl_address, alt_u32 data)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`write_address`—the base address of the FIFO write agent `ctrl_address`—the base address of the FIFO control agent to read the status `data`—the packet status information to write to address offset 1 of the Avalon interface. See the **Avalon Interface Specifications** section for the ordering of the packet status information.
Returns:	Returns 0 (`ALTERA_AVALON_FIFO_OK`) if successful, `ALTERA_AVALON_FIFO_FULL` if unsuccessful.
Description:	Writes the packet status information to the `write_address`. Only valid when **Enable packet data** is on.

24.6.14. altera_avalon_fifo_read_fifo()

Prototype:	`int altera_avalon_fifo_read_fifo(alt_u32 read_address, alt_u32 ctrl_address)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`read_address`—the base address of the FIFO read agent `ctrl_address`—the base address of the FIFO control agent to read the status
Returns:	Returns the data from address offset 0, or 0 if the FIFO is empty.
Description:	Gets the data addressed by `read_address`.

24.6.15. altera_avalon_fifo_read_other_info()

Prototype:	`int altera_avalon_fifo_read_other_info(alt_u32 read_address)`
Thread-safe:	No.
Available from ISR:	No.
Include:	<**altera_avalon_fifo_regs.h**>, <**altera_avalon_fifo_utils.h**>
Parameters:	`read_address`—the base address of the FIFO read agent
Returns:	Returns the packet status information from address offset 1 of the Avalon interface. See the **Avalon Interface Specifications** section for the ordering of the packet status information.
Description:	Reads the packet status information from the specified `read_address`. Only valid when **Enable packet data** is on.

24.7. Intel FPGA Avalon FIFO Memory Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2019.07.16	19.1	• Corrected the IP core name: from OnChip FIFO Memory Core to Intel FPGA Avalon FIFO Memory Core. • Corrected the following signal names: — `clk_in` to `wrclock` — `reset_in` to `reset_n` — `clk_out` to `rdclock` • Updated the description of `ctrl_address` in the *Intel FPGA Avalon FIFO Memory API* section.
2018.09.24	18.1	Added a new section: *Interface Signals*.
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
July 2014	2014.07.24	Removed mention of Platform Designer, updated to Platform Designer
December 2010	v10.1.0	Removed the “Device Support”, “Instantiating the Core in Platform Designer”, and “Referenced Documents” sections.
July 2010	v10.0.0	Revised the description of the memory map.
November 2009	v9.1.0	Added description to the core overview.
March 2009	v9.0.0	Updated the description of the function altera_avalon_fifo_read_status().
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	v8.0.0	No change from previous release.

25. On-Chip Memory (RAM and ROM) Core

25.1. Core Overview

Intel FPGAs include on-chip memory blocks that can be used as RAM or ROM in Platform Designer systems. On-chip memory has the following benefits for Platform Designer systems:

- On-chip memory has fast access time, compared to off-chip memory.
- Platform Designer automatically instantiates on-chip memory inside the Platform Designer system, so you do not have to make any manual connections.
- Certain memory blocks can have initialized contents when the FPGA powers up. This feature is useful, for example, for storing data constants or processor boot code.
- On-chip memories support dual port accesses, allowing two host to access the same memory concurrently.

25.2. Component-Level Design for On-Chip Memory

In Platform Designer you instantiate on-chip memory by clicking On-chip Memory (RAM or ROM) from the list of available components. The configuration window for the On-chip Memory (RAM or ROM) component has the following options: **Memory type**, **Size**, and **Read latency**.

25.2.1. Memory Type

This options defines the structure of the on-chip memory:

- RAM (writable)—This setting creates a readable and writable memory.
- ROM (read only)—This setting creates a read-only memory.
- Dual-port access—This setting creates a memory component with two agents, which allows two hosts to access the memory simultaneously.

  Note: The memory component operates under true dual-port mode where both agent ports have address ports for read or write operations. If two hosts access the same address simultaneously in a dual-port memory undefined results will occur. Concurrent accesses are only a problem for two writes. A read and write to the same location will read out the old data and store the new data.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

- Single clock operation—Single clock operation setting creates single clock source to clock both agents port. If single clock operation is not selected, each of the two agents port is clocked by different clock sources.

  Note: For Intel Stratix 10 devices, only single clock operation is supported.
- Read During Write Mode—This setting determines what the output data of the memory should be when a simultaneous read and write to the same memory location occurs.
- Block type—This setting directs the Intel Quartus Prime software to use a specific type of memory block when fitting the on-chip memory in the FPGA.

  Note: The MRAM blocks do not allow the contents to be initialized during power up. The M512s memory type does not support dual-port mode where both ports support both reads and writes.

Because of the constraints on some memory types, it is frequently best to use the **Auto** setting. **Auto** allows the Intel Quartus Prime software to choose a type and the other settings direct the Intel Quartus Prime software to select a particular type.

25.2.2. Size

This options defines the size and width of the memory.

- Enable different width for Dual-port Access—Different width for dual-port access status.

  Note: A different width for dual-port access is not supported for Intel Stratix 10 devices.
- Agent S1 Data width—This setting determines the data width of the memory. The available choices are 8, 16, 32, 64, 128, 256, 512, or 1024 bits. Assign Data width to match the width of the host that accesses this memory the most frequently or has the most critical throughput requirements. For example, if you are connecting the on-chip memory to the data host of a Nios II processor, you should set the data width of the on-chip memory to 32 bits, the same as the data-width of the Nios II data host. Otherwise, the access latency could be longer than one cycle because the Avalon interconnect fabric performs width translation.
- Total memory size—This setting determines the total size of the on-chip memory block. The total memory size must be less than the available memory in the target FPGA.

  The IP parameter editor accepts characters **k** and **m** to specify the memory size in kilobytes and megabytes respectively. For example: if you enter 1k, it will automatically resolve to its equivalent bytes which is 1024 bytes in this case.
- Minimize memory block usage (may impact fmax)—Minimize memory block usage (may impact fmax)—This option is only available for devices that include M4K memory blocks. If selected, the Intel Quartus Prime software divides the memory by depth rather than width, so that fewer memory blocks are used. This change may decrease fmax.

25.2.3. Read Latency

On-chip memory components use synchronous, pipelined Avalon-MM agents. Pipelined access improves fMAX performance, but also adds latency cycles when reading the memory. The Read latency option allows you to specify either one or two cycles of read latency required to access data. If the Dual-port access setting is turned on, you

can specify a different read latency for each agent. When you have dual-port memory in your system you can specify different clock frequencies for the ports. You specify this on the System Contents tab in Platform Designer.

25.2.4. ROM/RAM Memory Protection

This setting if enabled, creates additional reset request port for memory protection during reset. This additional reset input port is used to gate off the clock to the memory.

25.2.5. ECC Parameter

This setting if enabled, extends the data width to support ECC bits. It does not instantiate any ECC encoder or decoder logic within this component.

25.2.6. Memory Initialization

The memory initialization parameter section contains the following options:

- Initialize memory content—Option for user to enable memory content initialization.
- Enable non-default initialization file—You can specify your own initialization file by selecting Enable non-default initialization file. This option allows the file you specify to be used to initialize the memory in place of the default initialization file created by Platform Designer.
- Enable Partial Reconfiguration Initialization Mode—This setting if enabled, automatically instantiates logic to support Partial Reconfiguration use cases for initialized memory.
- Enable In-System Memory Content Editor Feature—Enables a JTAG interface used to read and write to the RAM while it is operating. You can use this interface to update or read the contents of the memory from your host PC.

25.3. Platform Designer System-Level Design for On-Chip Memory

There are few Platform Designer system-level design considerations for on-chip memories. See “Platform Designer System-Level Design”.

When generating a new system, Platform Designer creates a blank initialization file in the Intel Quartus Prime project directory for each on-chip memory that can power up with initialized contents. The name of this file is `<name of memory component>.hex`.

25.4. Simulation for On-Chip Memory

At system generation time, Platform Designer generates a simulation model for the on-chip memory. This model is embedded inside the Platform Designer system, and there are no user-configurable options for the simulation testbench.

You can provide memory initialization contents for simulation in the file `<Intel Quartus Prime project directory>/<Platform Designer system name>_sim/<Memory component name>`**.hex**.

25.5. Intel Quartus Prime Project-Level Design for On-Chip Memory

The on-chip memory is embedded inside the Platform Designer system, and there are no signals to connect to the Intel Quartus Prime project.

To provide memory initialization contents, you must fill in the file <name of memory component>**.hex**. The Intel Quartus Prime software recognizes this file during design compilation and incorporates the contents into the configuration files for the FPGA.

Note: For the memory to be initialized, you then must compile the hardware in the Intel Quartus Prime software for the SRAM Object File (**.sof**) to pick up the memory initialization files. All memory types with the exception of MRAMs support this feature.

25.6. Board-Level Design for On-Chip Memory

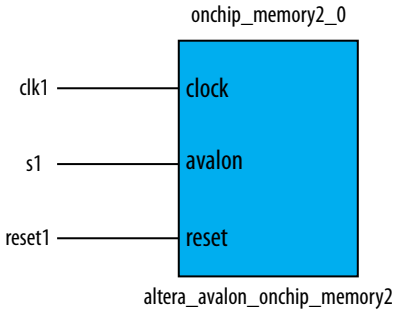
The on-chip memory is embedded inside the Platform Designer system, and there is nothing to connect at the board level.

25.7. Example Design with On-Chip Memory

This section demonstrates adding a 4 KByte on-chip RAM to the example design. This memory uses a single agent interface with a read latency of one cycle.

For demonstration purposes, the figure below shows the result of generating the Platform Designer system at this stage. In a normal design flow, you generate the system only after adding all system components.

Figure 77. Platform Designer System with On-Chip Memory



Because the on-chip memory is contained entirely within the Platform Designer system, Platform Designer `memory_system` has no I/O signals associated with `onchip_ram`. Therefore, you do not need to make any Intel Quartus Prime project connections or assignments for the on-chip RAM, and there are no board-level considerations.

25.8. On-Chip Memory (RAM and ROM) Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
May 2017	2017.05.08	Initial release

26. Optrex 16207 LCD Controller Core

Attention: The Optrex 16207 LCD Controller Core is part of a product obsolescence and support discontinuation schedule. For new design, Intel recommends that you use other IPs with equivalent functions. To see a list of available IPs, refer to the Intel FPGA IP Portfolio page.

26.1. Core Overview

The Optrex 16207 LCD controller core with Avalon Interface (LCD controller core) provides the hardware interface and software driver required for a Nios II processor to display characters on an Optrex 16207 (or equivalent) 16×2-character LCD panel. Device drivers are provided in the HAL system library for the Nios II processor. Nios II programs access the LCD controller as a character mode device using ANSI C standard library routines, such as `printf()`. The LCD controller is Platform Designer-ready, and integrates easily into any Platform Designer-generated system.

The Nios II Embedded Design Suite (EDS) includes an Optrex LCD module and provide several ready-made example designs that display text on the Optrex 16207 via the LCD controller.

For details about the Optrex 16207 LCD module, see the manufacturer's *Dot Matrix Character LCD Module User's Manual* available online.

26.2. Functional Description

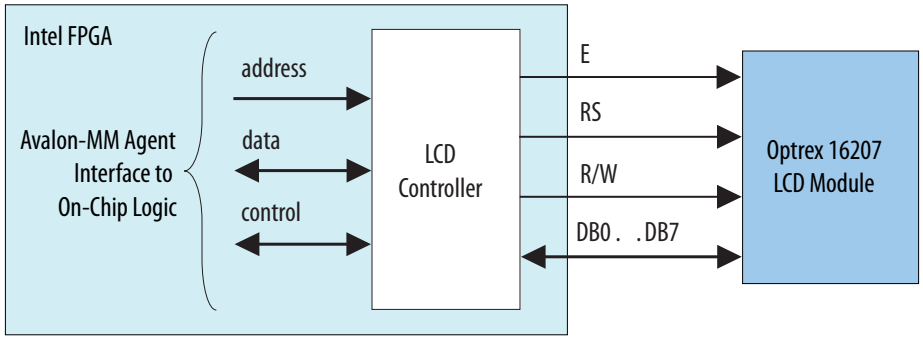
The LCD controller core consists of two user-visible components:

- Eleven signals that connect to pins on the Optrex 16207 LCD panel—These signals are defined in the Optrex 16207 data sheet.
  - E—Enable (output)
  - RS—Register Select (output)
  - R/W—Read or Write (output)
  - DB0 through DB7—Data Bus (bidirectional)
- An Avalon Memory-Mapped (Avalon-MM) agent interface that provides access to 4 registers.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Figure 78. LCD Controller Block Diagram



26.3. Software Programming Model

This section describes the software programming model for the LCD controller.

26.3.1. HAL System Library Support

Intel provides HAL system library drivers for the Nios II processor that enable you to access the LCD controller using the ANSI C standard library functions. The Intel-provided drivers integrate into the HAL system library for Nios II systems. The LCD driver is a standard character-mode device, as described in the **Nios II Software Developer's Handbook**. Therefore, using `printf()` is the easiest way to write characters to the display.

The LCD driver requires that the HAL system library include the system clock driver.

26.3.2. Displaying Characters on the LCD

The driver implements VT100 terminal-like behavior on a miniature scale for the 16×2 screen. Characters written to the LCD controller are stored to an 80-column × 2-row buffer maintained by the driver. As characters are written, the cursor position is updated. Visible characters move the cursor position to the right. Any visible characters written to the right of the buffer are discarded. The line feed character (`\n`) moves the cursor down one line and to the left-most column.

The buffer is scrolled up as soon as a printable character is written onto the line below the bottom of the buffer. Rows do not scroll as soon as the cursor moves down to allow the maximum useful information in the buffer to be displayed.

If the visible characters in the buffer fit on the display, all characters are displayed. If the buffer is wider than the display, the display scrolls horizontally to display all the characters. Different lines scroll at different speeds, depending on the number of characters in each line of the buffer.

The LCD driver supports a small subset of ANSI and VT100 escape sequences that can be used to control the cursor position, and clear the display as shown below.

Table 271. Escape Sequence Supported by the LCD Controller

Sequence	Meaning
BS (\b)	Moves the cursor to the left by one character.
CR (\r)	Moves the cursor to the start of the current line.
LF (\n)	Moves the cursor to the start of the line and move it down one line.
ESC ((\x1B)	Starts a VT100 control sequence.
ESC [<y> ; <x> H	Moves the cursor to the y, x position specified – positions are counted from the top left which is 1;1.
ESC [K	Clears from current cursor position to end of line.
ESC [2 J	Clears the whole screen.

The LCD controller is an output-only device. Therefore, attempts to read from it returns immediately indicating that no characters have been received.

The LCD controller drivers are not included in the system library when the **Reduced device drivers** option is enabled for the system library. If you want to use the LCD controller while using small drivers for other devices, add the preprocessor option—DALT_USE_LCD_16207 to the preprocessor options.

26.3.3. Software Files

The LCD controller is accompanied by the following software files. These files define the low-level interface to the hardware and provide the HAL drivers. Application developers should not modify these files.

- `altera_avalon_lcd_16207_regs.h` — This file defines the core's register map, providing symbolic constants to access the low-level hardware.
- `altera_avalon_lcd_16207.h`, `altera_avalon_lcd_16207.c` — These files implement the LCD controller device drivers for the HAL system library.

26.3.4. Register Map

The HAL device drivers make it unnecessary for you to access the registers directly. Therefore, Intel does not publish details about the register map. For more information, the `altera_avalon_lcd_16207_regs.h` file describes the register map, and the *Dot Matrix Character LCD Module User's Manual* from Optrex describes the register usage.

26.3.5. Interrupt Behavior

The LCD controller does not generate interrupts. However, the LCD driver's text scrolling feature relies on the HAL system clock driver, which uses interrupts for timing purposes.

26.4. Optrex 16207 LCD Controller Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2021.03.29	21.1	Added product obsolescence and support discontinuation notice.
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
July 2014	2014.07.24	Removed mention of SOPC Builder, updated to Platform Designer
December 2010	v10.1.0	Removed the “Device Support”, “Instantiating the Core in SOPC Builder”, and “Referenced Documents” sections.
July 2010	v10.0.0	No change from previous release.
November 2009	v9.1.0	No change from previous release.
March 2009	v9.0.0	No change from previous release.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	v8.0.0	No change from previous release.

27. PIO Core

27.1. Core Overview

The parallel input/output (PIO) core with Avalon interface provides a memory-mapped interface between an Avalon Memory-Mapped (Avalon-MM) agent port and general-purpose I/O ports. The I/O ports connect either to on-chip user logic, or to I/O pins that connect to devices external to the FPGA.

The PIO core provides easy I/O access to user logic or external devices in situations where a "bit banging" approach is sufficient. Some example uses are:

- Controlling LEDs
- Acquiring data from switches
- Controlling display devices
- Configuring and communicating with off-chip devices, such as application-specific standard products (ASSP)

  The PIO core interrupt request (IRQ) output can assert an interrupt based on input signals.

Note: This IP core supports Intel eASIC N5X devices.

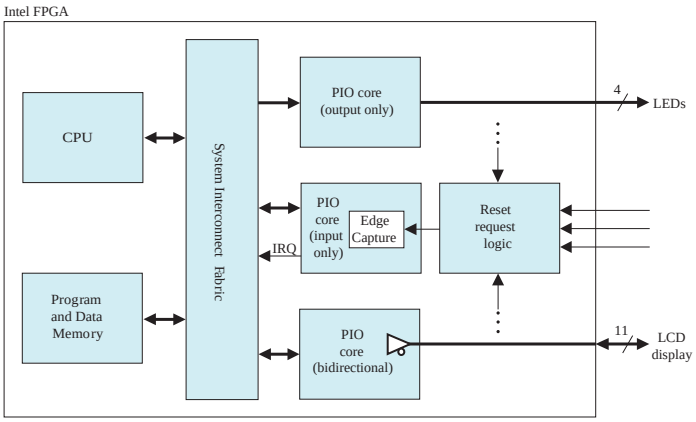
27.2. Functional Description

Each PIO core can provide up to 32 I/O ports. An intelligent host such as a microprocessor controls the PIO ports by reading and writing the register-mapped Avalon-MM interface. Under control of the host, the PIO core captures data on its inputs and drives data to its outputs. When the PIO ports are connected directly to I/O pins, the host can tristate the pins by writing control registers in the PIO core. The example below shows a processor-based system that uses multiple PIO cores to drive LEDs, capture edges from on-chip reset-request control logic, and control an off-chip LCD display.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Figure 79. System Using Multiple PIO Cores



When integrated into an Platform Designer-generated system, the PIO core has two user-visible features:

- A memory-mapped register space with four registers: `data`, `direction`, `interruptmask`, and `edgecapture`
- 1 to 32 I/O ports

  The I/O ports can be connected to logic inside the FPGA, or to device pins that connect to off-chip devices. The registers provide an interface to the I/O ports via the Avalon-MM interface. See **Register Map for the PIO Core** table for a description of the registers.

27.2.1. Data Input and Output

The PIO core I/O ports can connect to either on-chip or off-chip logic. The core can be configured with inputs only, outputs only, or both inputs and outputs. If the core is used to control bidirectional I/O pins on the device, the core provides a bidirectional mode with tristate control.

The hardware logic is separate for reading and writing the data register. Reading the data register returns the value present on the input ports (if present). Writing data affects the value driven to the output ports (if present). These ports are independent; reading the data register does not return previously-written data.

27.2.2. Edge Capture

The PIO core can be configured to capture edges on its input ports. It can capture low-to-high transitions, high-to-low transitions, or both. Whenever an input detects an edge, the condition is indicated in the `edgecapture` register. The types of edges detected is specified during IP generation in the Platform Designer, and cannot be changed via the registers.

27.2.3. IRQ Generation

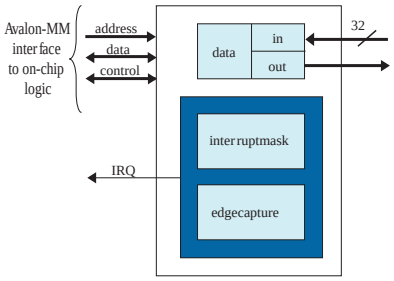
The PIO core can be configured to generate an IRQ on certain input conditions. The IRQ conditions can be either:

- Level-sensitive—The PIO core hardware can detect a high level. A NOT gate can be inserted external to the core to provide negative sensitivity.
- Edge-sensitive—The core's edge capture configuration determines which type of edge causes an IRQ

  Interrupts are individually maskable for each input port. The interrupt mask determines which input port can generate interrupts.

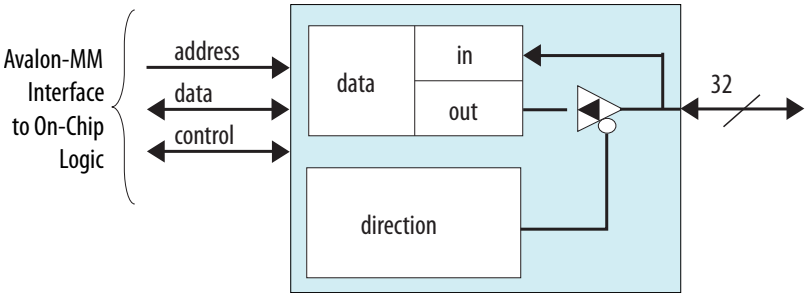
27.3. Example Configurations

Figure 80. PIO Core with Input Ports, Output Ports, and IRQ Support



The block diagram below shows the PIO core configured in bidirectional mode, without support for IRQs.

Figure 81. PIO Cores with Bidirectional Ports



27.3.1. Avalon-MM Interface

The PIO core's Avalon-MM interface consists of a single Avalon-MM agent port. The agent port is capable of fundamental Avalon-MM read and write transfers. The Avalon-MM agent port provides an IRQ output so that the core can assert interrupts.

27.4. Configuration

The following sections describe the available configuration options.

27.4.1. Basic Settings

The **Basic Settings** page allows you to specify the width, direction and reset value of the I/O ports.

27.4.1.1. Width

The width of the I/O ports can be set to any integer value between 1 and 32.

27.4.1.2. Direction

You can set the port direction to one of the options shown below.

Table 272. Direction Settings

Setting	Description
Bidirectional (tristate) ports	In this mode, each PIO bit shares one device pin for driving and capturing data. The direction of each pin is individually selectable. To tristate an FPGA I/O pin, set the direction to input.
Input ports only	In this mode the PIO ports can capture input only.
Output ports only	In this mode the PIO ports can drive output only.
Both input and output ports	In this mode, the input and output ports buses are separate, unidirectional buses of n bits wide.

27.4.1.3. Output Port Reset Value

You can specify the reset value of the output ports. The range of legal values depends on the port width.

27.4.1.4. Output Register

The option **Enable individual bit set/clear output register** allows you to set or clear individual bits of the output port. When this option is turned on, two additional registers—`outset` and `outclear`—are implemented. You can use these registers to specify the output bit to set and clear.

27.4.2. Input Options

The **Input Options** page allows you to specify edge-capture and IRQ generation settings. The **Input Options** page is not available when **Output ports only** is selected on the **Basic Settings** page.

27.4.2.1. Edge Capture Register

Turn on **Synchronously capture** to include the edge capture register, `edgecapture`, in the core. The edge capture register allows the core to detect and generate an optional interrupt when an edge of the specified type occurs on an input port. The user must further specify the following features:

- Select the type of edge to detect:
  - **Rising Edge**
  - **Falling Edge**
  - **Either Edge**
- Turn on **Enable bit-clearing for edge capture register** to clear individual bit in the edge capture register. To clear a given bit, write 1 to the bit in the edge capture register.

27.4.2.2. Interrupt

Turn on **Generate IRQ** to assert an IRQ output when a specified event occurs on input ports. The user must further specify the cause of an IRQ event:

- **Level—** The core generates an IRQ whenever a specific input is high and interrupts are enabled for that input in the `interruptmask` register.
- **Edge**— The core generates an IRQ whenever a specific bit in the edge capture register is high and interrupts are enabled for that bit in the `interruptmask` register.

  When **Generate IRQ** is off, the `interruptmask` register does not exist.

27.4.3. Simulation

The **Simulation** page allows you to specify the value of the input ports during simulation. Turn on **Hardwire PIO inputs in test bench** to set the PIO input ports to a certain value in the testbench, and specify the value in **Drive inputs to** field.

27.5. Software Programming Model

This section describes the software programming model for the PIO core, including the register map and software constructs used to access the hardware. For Nios II processor users, Intel provides the HAL system library header file that defines the PIO core registers. The PIO core does not match the generic device model categories supported by the HAL, so it cannot be accessed via the HAL API or the ANSI C standard library.

The Nios II Embedded Design Suite (EDS) provides several example designs that demonstrate usage of the PIO core. In particular, the `count_binary.c` example uses the PIO core to drive LEDs, and detect button presses using PIO edge-detect interrupts.

27.5.1. Software Files

The PIO core is accompanied by one software file, `altera_avalon_pio_regs.h`. This file defines the core's register map, providing symbolic constants to access the low-level hardware.

27.5.2. Register Map

An Avalon-MM host peripheral, such as a CPU, controls and communicates with the PIO core via the four 32-bit registers, shown below. The table assumes that the PIO core's I/O ports are configured to a width of n bits.

Table 273. Register Map for the PIO Core

Offset	Register Name		R/W	(n-1)	...	2	1	0
0	`data`	read access	R	Data value currently on PIO inputs				
		write access	W	New value to drive on PIO outputs				
1	`direction` (1)		R/W	Individual direction control for each I/O port. A value of 0 sets the direction to input; 1 sets the direction to output.				

continued...

Offset	Register Name	R/W	(n-1)	...	2	1	0
2	interruptmask (1)	R/W	IRQ enable/disable for each input port. Setting a bit to 1 enables interrupts for the corresponding port.				
3	edgecapture (1) , (2)	R/W	Edge detection for each input port.				
4	outset	W	Specifies which bit of the output port to set. Outset value is not stored into a physical register in the IP core. Hence it's value is not reserve for future use.				
5	outclear	W	Specifies which output bit to clear. Outclear value is not stored into a physical register in the IP core. Hence it's value is not reserve for future use.				

Note :

1. This register may not exist, depending on the hardware configuration. If a register is not present, reading the register returns an undefined value, and writing the register has no effect.
2. If the option **Enable bit-clearing for edge capture register** is turned off, writing any value to the edgecapture register clears all bits in the register. Otherwise, writing a 1 to a particular bit in the register clears only that bit.

27.5.2.1. data Register

Reading from data returns the value present at the input ports if the PIO core hardware is configured to input, or inout mode only. If the PIO core hardware is configured to output-only mode, reading from the data register returns the value present at the output ports. Whereas, if the PIO core hardware is configured to bidirectional mode, reading from data register returns value depending on the direction register value, setting to 1 returns value present at the output ports, setting to 0 returns undefined value.

Writing to data stores the value to a register that drives the output ports. If the PIO core hardware is configured in input-only mode, writing to data has no effect. If the PIO core hardware is in bidirectional mode, the registered value appears on an output port only when the corresponding bit in the direction register is set to 1 (output).

27.5.2.2. direction Register

The direction register controls the data direction for each PIO port, assuming the port is bidirectional. When bit n in direction is set to 1, port n drives out the value in the corresponding bit of the data register.

The direction register only exists when the PIO core hardware is configured in bidirectional mode. In input-only, output-only and inout mode, the direction register does not exist. In this case, reading direction returns an undefined value, writing direction has no effect. The mode (input, output, inout or bidirectional) is specified at system generation time, and cannot be changed at runtime.

After reset, all direction register bits are 0, so that all bidirectional I/O ports are configured as inputs. If those PIO ports are connected to device pins, the pins are held in a high-impedance state. In bi-directional mode, you will need to write to the direction register to change the direction of the PIO port (0-input, 1-output).

27.5.2.3. interruptmask Register

Setting a bit in the interruptmask register to 1 enables interrupts for the corresponding PIO input port. Interrupt behavior depends on the hardware configuration of the PIO core. See the **Interrupt Behavior** section.

The `interruptmask` register only exists when the hardware is configured to generate IRQs. If the core cannot generate IRQs, reading `interruptmask` returns an undefined value, and writing to `interruptmask` has no effect.

After reset, all bits of `interruptmask` are zero, so that interrupts are disabled for all PIO ports.

27.5.2.4. edgecapture Register

Bit *n* in the `edgecapture` register is set to 1 whenever an edge is detected on input port n. An Avalon-MM host peripheral can read the `edgecapture` register to determine if an edge has occurred on any of the PIO input ports. If the edge capture register bit has been previously set, `in_port` toggling activity will not change value.

If the option Enable bit-clearing for the edge capture register is turned off, writing any value to the `edgecapture` register clears all bits in the register. Otherwise, writing a 1 to a particular bit in the register clears only that bit.

The type of edge(s) to detect is specified during IP generation in Platform Designer. The `edgecapture` register only exists when the hardware is configured to capture edges. If the core is not configured to capture edges, reading from `edgecapture` returns an undefined value, and writing to `edgecapture` has no effect.

27.5.2.5. outset and outclear Register

You can use the `outset` and `outclear` registers to set and clear individual bits of the output port. For example, to set bit 6 of the output port, write `0x40` to the `outset` register. Writing `0x08` to the `outclear` register clears bit 3 of the output port.

These registers are only present when the option **Enable individual bit set/clear output register** is turned on. Outset and `outclear` registers are not physical registers inside the IP core, hence the output port value will only be affected by the current update outset value or current update outclear value only.

27.5.3. Interrupt Behavior

The PIO core outputs a single IRQ signal that can connect to any host peripheral in the system. The host can read either the `data` register or the `edgecapture` register to determine which input port caused the interrupt.

When the hardware is configured for level-sensitive interrupts, the IRQ is asserted whenever corresponding bits in the `data` and `interruptmask` registers are 1. When the hardware is configured for edge-sensitive interrupts, the IRQ is asserted whenever corresponding bits in the `edgecapture` and `interruptmask` registers are 1. The IRQ remains asserted until explicitly acknowledged by disabling the appropriate bit(s) in `interruptmask`, or by writing to `edgecapture`.

27.5.4. Software Files

The PIO core is accompanied by the following software file. This file provide low-level access to the hardware. Application developers should not modify the file.

- `altera_avalon_pio_regs.h`—This file defines the core's register map, providing symbolic constants to access the low-level hardware. The symbols in this file are used by device driver functions.

27.6. PIO Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2021.03.29	21.1	Added support for Intel eASIC N5X devices.
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
December 2015	2015.12.16	Updated "edgecapture Register" section
June 2015	2015.06.12	• Updated "Register Map" section • Updated "data Register" section • Updated "direction Register" section • Updated "outset and outclear Register" section
July 2014	2014.07.24	Removed mention of SOPC Builder, updated to Platform Designer
December 2013	v13.1.0	Updated note (2) in Register map for PIO Core Table
December 2010	v10.1.0	Removed the “Device Support”, “Instantiating the Core in SOPC Builder”, and “Referenced Documents” sections
July 2010	v10.0.0	No change from previous release
November 2009	v9.1.0	No change from previous release
March 2009	v9.0.0	Added a section on new registers, `outset` and `outclear`
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. Added the description for **Output Port Reset Value** and **Simulation** parameters
May 2008	v8.0.0	No change from previous release

28. PLL Cores

28.1. Core Overview

The PLL cores, Avalon ALTPLL and PLL, provide a means of accessing the dedicated on-chip PLL circuitry in the Intel Stratix (except Stratix V), and Cyclone series FPGAs. Both cores are a component wrapper around the Intel FPGA ALTPLL IP core.

The PLL core is scheduled for product obsolescence and discontinued support. Therefore, Intel recommends that you use the Avalon ALTPLL core in your designs.

The core takes an Platform Designer system clock as its input and generates PLL output clocks locked to that reference clock.

The PLL cores support the following features:

- All PLL features provided by Intel FPGA ALTPLL IP core. The exact feature set depends on the device family.
- Access to status and control signals via Avalon Memory-Mapped (Avalon-MM) registers or top-level signals on the Platform Designer system module.
- Dynamic phase reconfiguration in Stratix III and Stratix IV device families.

  The PLL output clocks are made available in two ways:
- As sources to system-wide clocks in your Platform Designer system.
- As output signals on your Platform Designer system module.

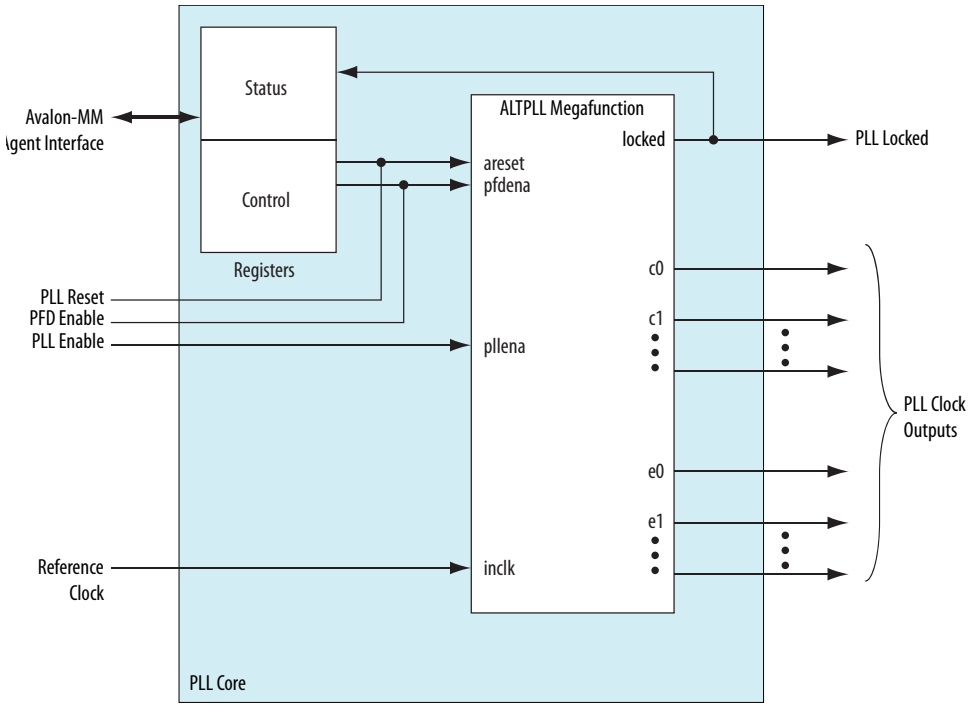
  For details about the ALTPLL IP core, refer to the ALTPLL IP Core User Guide.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

28.2. Functional Description

Figure 82. PLL Core Block Diagram



28.2.1. ALTPLL IP Core

The PLL cores consist of an ALTPLL IP core instantiation and an Avalon-MM agent interface. This interface can optionally provide access to status and control registers within the cores. The ALTPLL IP core takes an Platform Designer system clock as its reference, and generates one or more phase-locked loop output clocks.

28.2.2. Clock Outputs

Depending on the target device family, the ALTPLL IP core can produce two types of output clock:

- internal (c)—clock outputs that can drive logic either inside or outside the Platform Designer system module. Internal clock outputs can also be mapped to top-level FPGA pins. Internal clock outputs are available on all device families.
- external (e)—clock outputs that can only drive dedicated FPGA pins. They cannot be used as on-chip clock sources. External clock outputs are not available on all device families.

  The Avalon ALTPLL core, however, does not differentiate the internal and external clock outputs and allows the external clock outputs to be used as on-chip clock sources.

  To determine the exact number and type of output clocks available on your target device, refer to the ALTPLL IP Core User Guide.

28.2.3. PLL Status and Control Signals

Depending on how the ALTPLL IP core is parameterized, there can be a variable number of status and control signals. You can choose to export certain status and control signals to the top-level Platform Designer system module. Alternatively, Avalon-MM registers can provide access to the signals. Any status or control signals which are not mapped to registers are exported to the top-level module. For details, refer to the **Instantiating the Avalon ALTPLL Core**.

28.2.4. System Reset Considerations

At FPGA configuration, the PLL cores reset automatically. PLL-specific reset circuitry guarantees that the PLL locks before releasing reset for the overall Platform Designer system module.

Resetting the PLL resets the entire Platform Designer system module.

28.3. Instantiating the Avalon ALTPLL Core

When you instantiate the Avalon ALTPLL core, the MegaWizard Plug-In Manager is automatically launched for you to parameterize the ALTPLL IP core. There are no additional parameters that you can configure in Platform Designer.

The `pfdena` signal of the ALTPLL IP core is not exported to the top level of the Platform Designer module. You can drive this port by writing to the PFDENA bit in the `control` register.

The `locked`, `pllena`/`extclkena`, and `areset` signals of the IP core are always exported to the top level of the Platform Designer module. You can read the `locked` signal and reset the core by manipulating respective bits in the registers. See the **Register Definitions and Bit List** section for more information on the registers.

For details about using the ALTPLL MegaWizard Plug-In Manager, refer to the ALTPLL IP Core User Guide.

28.4. Instantiating the PLL Core

This section describes the options available in the MegaWizard™ interface for the PLL core in Platform Designer.

PLL Settings Page

The **PLL Settings** page contains a button that launches the ALTPLL MegaWizard Plug-In Manager. Use the MegaWizard Plug-In Manager to parameterize the ALTPLL IP core. The set of available parameters depends on the target device family.

You cannot click **Finish** in the PLL wizard nor configure the PLL interface until you parameterize the ALTPLL IP core.

Interface Page

The **Interface** page configures the access modes for the optional advanced PLL status and control signals.

For each advanced signal present on the ALTPLL IP core, you can select one of the following access modes:

- **Export**—Exports the signal to the top level of the Platform Designer system module.
- **Register**—Maps the signal to a bit in a status or control register.

  The advanced signals are optional. If you choose not to create any of them in the ALTPLL MegaWizard Plug-In, the PLL's default behavior is as shown in below.

  You can specify the access mode for the advanced signals shown in below. The ALTPLL core signals, not displayed in this table, are automatically exported to the top level of the Platform Designer system module.

Table 274. ALTPLL Advanced Signal

ALTPLL Name	Input / Output	Avalon-MM PLL Wizard Name	Default Behavior	Description
`areset`	input	PLL Reset Input	The PLL is reset only at device configuration.	This signal resets the entire Platform Designer system module, and restores the PLL to its initial settings.
`pllena`	input	PLL Enable Input	The PLL is enabled.	This signal enables the PLL. `pllena` is always exported.
`pfdena`	input	PFD Enable Input	The phase-frequency detector is enabled.	This signal enables the phase-frequency detector in the PLL, allowing it to lock on to changes in the clock reference.
`locked`	output	PLL Locked Output	—	This signal is asserted when the PLL is locked to the input clock.

Asserting `areset` resets the entire Platform Designer system module, not just the PLL.

Finish

Click **Finish** to insert the PLL into the Platform Designer system. The PLL clock output(s) appear in the clock settings table on the Platform Designer **System Contents** tab.

If the PLL has external output clocks, they appear in the clock settings table like other clocks; however, you cannot use them to drive components within the Platform Designer system.

For details about using external output clocks, refer to the ALTPLL IP Core User Guide.

The Platform Designer automatically connects the PLL's reference clock input to the first available clock in the clock settings table.

If there is more than one Platform Designer system clock available, verify that the PLL is connected to the appropriate reference clock.

28.5. Hardware Simulation Considerations

The HDL files generated by Platform Designer for the PLL cores are suitable for both synthesis and simulation. The PLL cores support the standard Platform Designer simulation flow, so there are no special considerations for hardware simulation.

28.6. Register Definitions and Bit List

Device drivers can control and communicate with the cores through two memory-mapped registers, `status` and `control`. The width of these registers are 32 bits in the Avalon ALTPLL core but only 16 bits in the PLL core.

In the PLL core, the `status` and `control` bits shown in the PLL Cores Register map below are present only if they have been created in the ALTPLL MegaWizard Plug-In Manager, and set to **Register** on the **Interface** page in the PLL wizard. These registers are always created in the Avalon ALTPLL core.

Table 275. PLL Cores Register Map

Offset	Register Name	R/W	Bit Description												
			31/15 (2)	30	29	...	9	8	7	6	5	4	3	2	1	0
0	`status`	R/O	(1)												`phasedone`	`locked`
1	`control`	R/W	(1)												`pfdena`	`areset`
2	phase reconfig control	R/W	`phase`		(1)			`counter_number`								
3	—	—	Undefined													

Note :
1. Reserved. Read values are undefined. When writing, set reserved bits to zero.
2. The registers are 32-bit wide in the Avalon ALTPLL core and 16-bit wide in the PLL core.

28.6.1. Status Register

Embedded software can access the PLL status via the `status` register. Writing to `status` has no effect.

Table 276. Status Register Bits

Bit Number	Bit Name	Value after reset	Description
0	`locked` (2)	1	Connects to the `locked` signal on the ALTPLL IP core. The `locked` bit is high when valid clocks are present on the output of the PLL.
1	`phasedone` (2)	0	Connects to the `phasedone` signal on the ALTPLL IP core. The `phasedone` output of the ALTPLL is synchronized to the system clock.
2:15/31 (1)	—	—	Reserved. Read values are undefined.

Note :
1. The `status` register is 32-bit wide in the Avalon ALTPLL core and 16-bit wide in the PLL core.
2. Both the `locked` and `phasedone` outputs from the Avalon ALTPLL component are available as conduits and reflect the non-synchronized outputs from the ALTPLL.

28.6.2. Control Register

Embedded software can control the PLL via the `control` register. Software can also read back the status of control bits.

Table 277. Control Register Bits

Bit Number	Bit Name	Value after reset	Description
0	areset	0	Connects to the areset signal on the ALTPLL IP core. Writing a 1 to this bit initiates a PLL reset.
1	pfdena	1	Connects to the pfdena signal on the ALTPLL IP core. Writing a 0 to this bit disables the phase frequency detection.
2:15/31 (1)	—	—	Reserved. Read values are undefined. When writing, set reserved bits to zero.

Note :
1. The controlregister is 32-bit wide in the Avalon ALTPLL core and 16-bit wide in the PLL core.

28.6.3. Phase Reconfig Control Register

Embedded software can control the dynamic phase reconfiguration via the phase reconfig control register.

Table 278. Phase Reconfig Control Register Bits

Bit Number	Bit Name	Value after reset	Description
0:8	counter_number	—	A binary 9-bit representation of the counter that needs to be reconfigured. Refer to the Counter_Number Bits and Selection table for the counter selection.
9:29	—	—	Reserved. Read values are undefined. When writing, set reserved bits to zero.
30:31	phase (1)	—	01: Step up phase of counter_number 10: Step down phase of counter_number 00 and 11: No operation

Note :
1. Phase step up or down when set to 1 (only applicable to the Avalon ALTPLL core).

The table below lists the counter number and selection. For example, 100 000 000 selects counter C0 and 100 000 001 selects counter C1.

Table 279. Counter_Number Bits and Selection

Counter_Number [0:8]	Counter Selection
0 0000 0000	All output counters
0 0000 0001	M counter
> 0 0000 0001	Undefined
1 0000 0000	C0
1 0000 0001	C1
1 0000 0010	C2
...	...
1 0000 1000	C8
1 0000 1001	C9
> 1 0000 1001	Undefined

28.7. PLL Cores Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
December 2010	v10.1.0	Removed the "Device Support", "Instantiating the Core in SOPC Builder", and "Referenced Documents" sections.
July 2010	v10.0.0	No change from previous release.
November 2009	v9.1.0	Revised descriptions of register fields and bits.
March 2009	v9.0.0	Added information on the new Avalon ALTPLL core.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	v8.0.0	No change from previous release.

29. DMA Controller Core

29.1. Core Overview

The direct memory access (DMA) controller core with Avalon interface performs bulk data transfers, reading data from a source address range and writing the data to a different address range. An Avalon Memory-Mapped (Avalon-MM) host peripheral, such as a CPU, can offload memory transfer tasks to the DMA controller. While the DMA controller performs memory transfers, the host is free to perform other tasks in parallel.

The DMA controller transfers data as efficiently as possible, reading and writing data at the maximum pace allowed by the source or destination. The DMA controller is capable of performing Avalon transfers with flow control, enabling it to automatically transfer data to or from a slow peripheral with flow control (for example, UART), at the maximum pace allowed by the peripheral.

Instantiating the DMA controller in Platform Designer creates one agent port and two host ports. You must specify which agent peripherals can be accessed by the read and write host ports. Likewise, you must specify which other host peripheral(s) can access the DMA control port and initiate DMA transactions. The DMA controller does not export any signals to the top level of the system module.

Note: While instantiating the DMA controller in the hierarchical subsystem, add an Avalon-MM pipeline bridge in front of the exported host interface of the DMA controller in the subsystem. This will allow you to configure the bus width of the pipeline bridge to match the agent address bus.

For the Nios II processor, device drivers are provided in the HAL system library. See the **Software Programming Model** section for details of HAL support.

29.2. Functional Description

You can use the DMA controller to perform data transfers from a source address-space to a destination address-space. The controller has no concept of endianness and does not interpret the payload data. The concept of endianness only applies to a host that interprets payload data.

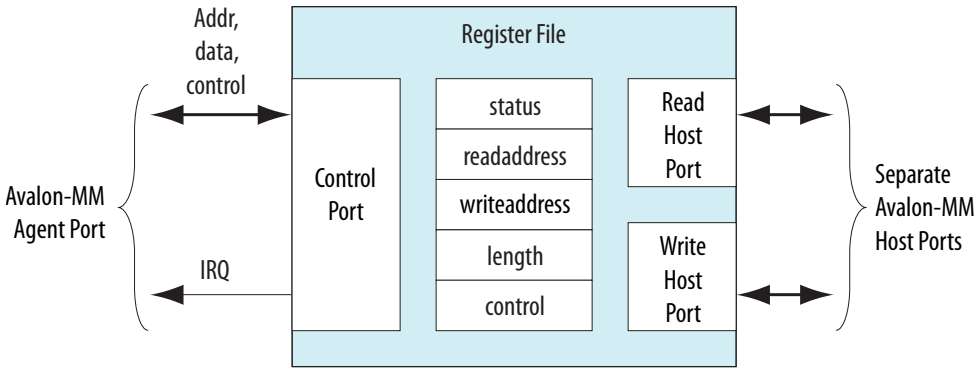
The source and destination may be either an Avalon-MM agent peripheral (for example, a constant address) or an address range in memory. The DMA controller can be used in conjunction with peripherals with flow control, which allows data transactions of fixed or variable length. The DMA controller can signal an interrupt request (IRQ) when a DMA transaction completes. A transaction is a sequence of one or more Avalon transfers initiated by the DMA controller core.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

The DMA controller has two Avalon-MM host ports—a host read port and a host write port—and one Avalon-MM agent port for controlling the DMA as shown in the figure below.

Figure 83. DMA Controller Block Diagram



A typical DMA transaction proceeds as follows:

1. A CPU prepares the DMA controller for a transaction by writing to the control port.
2. The CPU enables the DMA controller. The DMA controller then begins transferring data without additional intervention from the CPU. The DMA's host read port reads data from the read address, which may be a memory or a peripheral. The host write port writes the data to the destination address, which can also be a memory or peripheral. A shallow FIFO buffers data between the read and write ports.
3. The DMA transaction ends when a specified number of bytes are transferred (a fixed-length transaction) or an end-of-packet signal is asserted by either the sender or receiver (a variable-length transaction). At the end of the transaction, the DMA controller generates an interrupt request (IRQ) if it was configured by the CPU to do so.
4. During or after the transaction, the CPU can determine if a transaction is in progress, or if the transaction ended (and how) by examining the DMA controller's `status` register.

29.2.1. Setting Up DMA Transactions

An Avalon-MM host peripheral sets up and initiates DMA transactions by writing to registers via the control port. The Avalon-MM host programs the DMA engine using byte addresses which are byte aligned. The host peripheral configures the following options:

- Read (source) address location
- Write (destination) address location
- Size of the individual transfers: Byte (8-bit), halfword (16-bit), word (32-bit), doubleword (64-bit) or quadword (128-bit)

- Enable interrupt upon end of transaction
- Enable source or destination to end the DMA transaction with end-of-packet signal
- Specify whether source and destination are memory or peripheral

  The host peripheral then sets a bit in the `control` register to initiate the DMA transaction.

29.2.2. The Host Read and Write Ports

The DMA controller reads data from the source address through the host read port, and then writes to the destination address through the host write port. You program the DMA controller using byte addresses. Read and write start addresses should be aligned to the transfer size. For example, to transfer data words, if the start address is 0, the address will increment to 4, 8, and 12. For heterogeneous systems where a number of different agent devices are of different widths, the data width for read and write hosts matches the width of the widest data-width agent addressed by either the read or the write host. For bursting transfers, the burst length is set to the DMA transaction length with the appropriate unit conversion. For example, if a 32-bit data width DMA is programmed for a word transfer of 64 bytes, the length registered is programmed with 64 and the burst count port will be 16. If a 64-bit data width DMA is programmed for a doubleword transfer of 8 bytes, the length register is programmed with 8 and the burst count port will be 1.

There is a shallow FIFO buffer between the host read and write ports. The default depth is 2, which makes the write action depend on the data-available status of the FIFO, rather than on the status of the host read port.

Both the read and write host ports can perform Avalon transfers with flow control, which allows the agent peripheral to control the flow of data and terminate the DMA transaction.

For details about flow control in Avalon-MM data transfers and Avalon-MM peripherals, refer to Avalon Interface Specifications.

29.2.3. Addressing and Address Incrementing

When accessing memory, the read (or write) address increments by 1, 2, 4, 8, or 16 after each access, depending on the width of the data. On the other hand, a typical peripheral device (such as UART) has fixed register locations. In this case, the read/write address is held constant throughout the DMA transaction.

The rules for address incrementing are, in order of priority:

- If the `control` register's `RCON` (or `WCON`) bit is set, the read (or write) increment value is 0.
- Otherwise, the read and write increment values are set according to the transfer size specified in the control register, as shown below.

Table 280. Address Increment Values

Transfer Width	Increment
byte	1
halfword	2
word	4
doubleword	8
quadword	16

In systems with heterogeneous data widths, care must be taken to present the correct address or offset when configuring the DMA to access native-aligned agents. For example, in a system using a 32-bit Nios II processor and a 16-bit DMA, the base address for the UART `txdata` register must be divided by the dma_data_width/cpu_data_width—2 in this example.

29.3. Parameters

This section describes the parameters you can configure.

29.3.1. DMA Parameters (Basic)

The **DMA Parameters** page includes the following parameters.

Transfer Size

The parameter **Width of the DMA Length Register** specifies the minimum width of the DMA's transaction length register, which can be between 1 and 32. The `length` register determines the maximum number of transfers possible in a single DMA transaction.

By default, the length register is wide enough to span any of the agent peripherals hosted by the read or write ports. Overriding the length register may be necessary if the DMA host port (read or write) hosts only data peripherals, such as a UART. In this case, the address span of each agent is small, but a larger number of transfers may be desired per DMA transaction. A smaller transfer width usually results in a faster FPGA frequency for the DMA Controller Core.

Burst Transactions

When **Enable Burst Transfers** is turned on, the DMA controller performs burst transactions on its host read and write ports. The parameter **Maximum Burst Size** determines the maximum burst size allowed in a transaction.

In burst mode, the length of a transaction must not be longer than the configured maximum burst size. Otherwise, the transaction must be performed as multiple transactions.

FIFO Depth

The parameter **Data Transfer FIFO Depth** specifies the depth of the FIFO buffer used for data transfers. Intel recommends that you set the depth of the FIFO buffer to at least twice the maximum read latency of the agent interface connected to the read host port. A depth that is too low reduces transfer throughput.

FIFO Implementation

This option determines the implementation of the FIFO buffer between the host read and write ports. Select **Construct FIFO from Registers** to implement the FIFO using one register per storage bit. This option has a strong impact on logic utilization when the DMA controller's data width is large. See the **Advanced Options** section.

By default, the FIFO implementation uses the embedded memory blocks.

29.3.2. Advanced Options

The **Advanced Options** page includes the following parameters.

Allowed Transactions

You can choose the transfer datawidth(s) supported by the DMA controller hardware. The following datawidth options can be enabled or disabled:

- Byte
- Halfword (two bytes)
- Word (four bytes)
- Doubleword (eight bytes)
- Quadword (sixteen bytes)

  Disabling unnecessary transfer widths reduces the number of on-chip logic resources consumed by the DMA controller core. For example, if a system has both 16-bit and 32-bit memories, but the DMA controller transfers data to the 16-bit memory, 32-bit transfers could be disabled to conserve logic resources.

29.4. Software Programming Model

This section describes the programming model for the DMA controller, including the register map and software declarations to access the hardware. For Nios II processor users, Intel provides HAL system library drivers that enable you to access the DMA controller core using the HAL API for DMA devices.

29.4.1. HAL System Library Support

The Intel-provided driver implements a HAL DMA device driver that integrates into the HAL system library for Nios II systems. HAL users should access the DMA controller via the familiar HAL API, rather than accessing the registers directly.

If your program uses the HAL device driver to access the DMA controller, accessing the device registers directly interferes with the correct behavior of the driver.

The HAL DMA driver provides both ends of the DMA process; the driver registers itself as both a receive channel (`alt_dma_rxchan`) and a transmit channel (`alt_dma_txchan`). The *Nios II Software Developer's Handbook* provides complete details of the HAL system library and the usage of DMA devices.

ioctl() Operations

`ioctl()` operation requests are defined for both the receive and transmit channels, which allows you to control the hardware-dependent aspects of the DMA controller. Two `ioctl()` functions are defined for the receiver driver and the transmitter driver: `alt_dma_rxchan_ioctl()` and `alt_dma_txchan_ioctl()`. The table below lists the available operations. These are valid for both the transmit and receive channels.

Table 281. Operations for alt_dma_rxchan_ioctl() and alt_dma_txchan_ioctl()

Request	Meaning
ALT_DMA_SET_MODE_8	Transfers data in units of 8 bits. The parameter `arg` is ignored.
ALT_DMA_SET_MODE_16	Transfers data in units of 16 bits. The parameter `arg` is ignored.
ALT_DMA_SET_MODE_32	Transfers data in units of 32 bits. The parameter `arg` is ignored.
ALT_DMA_SET_MODE_64	Transfers data in units of 64 bits. The parameter `arg` is ignored.
ALT_DMA_SET_MODE_128	Transfers data in units of 128 bits. The parameter `arg` is ignored.
ALT_DMA_RX_ONLY_ON (1)	Sets a DMA receiver into streaming mode. In this case, data is read continuously from a single location. The parameter `arg` specifies the address to read from.
ALT_DMA_RX_ONLY_OFF (1)	Turns off streaming mode for a receive channel. The parameter `arg` is ignored.
ALT_DMA_TX_ONLY_ON (1)	Sets a DMA transmitter into streaming mode. In this case, data is written continuously to a single location. The parameter `arg` specifies the address to write to.
ALT_DMA_TX_ONLY_OFF (1)	Turns off streaming mode for a transmit channel. The parameter `arg` is ignored.

Notes :

1. These macro names changed in version 1.1 of the Nios II Embedded Design Suite (EDS). The old names (ALT_DMA_TX_STREAM_ON, ALT_DMA_TX_STREAM_OFF, ALT_DMA_RX_STREAM_ON, and ALT_DMA_RX_STREAM_OFF) are still valid, but new designs should use the new names.

Limitations

Currently the Intel-provided drivers do not support 64-bit and 128-bit DMA transactions.

This function is not thread safe. If you want to access the DMA controller from more than one thread then you should use a semaphore or mutex to ensure that only one thread is executing within this function at any time.

29.4.2. Software Files

The DMA controller is accompanied by the following software files. These files define the low-level interface to the hardware. Application developers should not modify these files.

- `altera_avalon_dma_regs.h`—This file defines the core's register map, providing symbolic constants to access the low-level hardware. The symbols in this file are used only by device driver functions.
- `altera_avalon_dma.h`, `altera_avalon_dma.c`—These files implement the DMA controller's device driver for the HAL system library.

29.4.3. Register Map

Programmers using the HAL API never access the DMA controller hardware directly via its registers. In general, the register map is only useful to programmers writing a device driver.

The Intel-provided HAL device driver accesses the device registers directly. If you are writing a device driver, and the HAL driver is active for the same device, your driver will conflict and fail to operate.

Device drivers control and communicate with the hardware through five memory-mapped 32-bit registers.

Table 282. DMA Controller Register Map

Offset	Register Name	Read / Write	31 13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	`status` (1)	RW	(2)									LEN	WEOP	REOP	BUSY	DONE
1	`readaddress`	RW	Read host start address													
2	`writeaddress`	RW	Write host start address													
3	`length`	RW	DMA transaction length (in bytes)													
4	—	—	Reserved (3)													
5	—	—	Reserved (3)													
6	`control`	RW	(2)	SOFTWARERESET	QUADWORD	DOUBLEWORD	WCON	RCON	LEEN	WEEN	REEN	I_EN	GO	WORD	HW	BYTE
7	—	—	Reserved (3)													

Notes :
1. Writing zero to the `status` register clears the LEN, WEOP, REOP, and DONE bits.
2. These bits are reserved. Read values are undefined. Write zero.
3. This register is reserved. Read values are undefined. The result of a write is undefined.

status Register

The `status` register consists of individual bits that indicate conditions inside the DMA controller. The `status` register can be read at any time. Reading the `status` register does not change its value.

Table 283. status Register Bits

Bit Number	Bit Name	Read/Write/Clear	Description
0	DONE	R/C	A DMA transaction is complete. The DONE bit is set to 1 when an end of packet condition is detected or the specified transaction length is completed. Write zero to the `status` register to clear the DONE bit.
1	BUSY	R	The BUSY bit is 1 when a DMA transaction is in progress.
2	REOP	R	The REOP bit is 1 when a transaction is completed due to an end-of-packet event on the read side.
3	WEOP	R	The WEOP bit is 1 when a transaction is completed due to an end of packet event on the write side.
4	LEN	R	The LEN bit is set to 1 when the length register decrements to zero.

readaddress Register

The `readaddress` register specifies the first location to be read in a DMA transaction. The `readaddress` register width is determined at system generation time. It is wide enough to address the full range of all agent ports hosted by the read port.

writeaddress Register

The `writeaddress` register specifies the first location to be written in a DMA transaction. The `writeaddress` register width is determined at system generation time. It is wide enough to address the full range of all agent ports hosted by the write port.

length Register

The `length` register specifies the number of bytes to be transferred from the read port to the write port. The `length` register is specified in bytes. For example, the value must be a multiple of 4 for word transfers, and a multiple of 2 for halfword transfers.

The `length` register is decremented as each data value is written by the write host port. When `length` reaches 0 the LEN bit is set. The `length` register does not decrement below 0.

The length register width is determined at system generation time. It is at least wide enough to span any of the agent ports hosted by the read or write host ports, and it can be made wider if necessary.

control Register

The control register is composed of individual bits that control the DMA's internal operation. The control register's value can be read at any time. The control register bits determine which, if any, conditions of the DMA transaction result in the end of a transaction and an interrupt request.

Table 284. Control Register Bits

Bit Number	Bit Name	Read/ Write/ Clear	Description
0	BYTE	RW	Specifies byte transfers.
1	HW	RW	Specifies halfword (16-bit) transfers.
2	WORD	RW	Specifies word (32-bit) transfers.
3	GO	RW	Enables DMA transaction. When the GO bit is set to 0 during idle stage (before execution starts), the DMA is prevented from executing transfers. When the GO bit is set to 1 during idle stage and the length register is non-zero, transfers occur. If go bit is de-asserted low before write transaction complete, done bit will never go high. It is advisable that GO bit is modified during idle stage (no execution happened) only.
4	I_EN	RW	Enables interrupt requests (IRQ). When the I_EN bit is 1, the DMA controller generates an IRQ when the status register's DONE bit is set to 1. IRQs are disabled when the I_EN bit is 0.
5	REEN	RW	Ends transaction on read-side end-of-packet. When the REEN bit is set to 1, a agent port with flow control on the read side may end the DMA transaction by asserting its end-of-packet signal. REEN bit should be set to 0.
6	WEEN	RW	Ends transaction on write-side end-of-packet. WEEN bit should be set to 0.
7	LEEN	RW	Ends transaction when the length register reaches zero. When this bit is 0, length reaching 0 does not cause a transaction to end. In this case, the DMA transaction must be terminated by an end-of-packet signal from either the read or write host port. LEEN bit should be set to 1.
8	RCON	RW	Reads from a constant address. When RCON is 0, the read address increments after every data transfer. This is the mechanism for the DMA controller to read a range of memory addresses. When RCON is 1, the read address does not increment. This is the mechanism for the DMA controller to read from a peripheral at a constant memory address. For details, see the **Addressing and Address Incrementing** section.
9	WCON	RW	Writes to a constant address. Similar to the RCON bit, when WCON is 0 the write address increments after every data transfer; when WCON is 1 the write address does not increment. For details, see **Addressing and Address Incrementing**.
10	DOUBLEWORD	RW	Specifies doubleword transfers.
11	QUADWORD	RW	Specifies quadword transfers.
12	SOFTWARERESET	RW	Software can reset the DMA engine by writing this bit to 1 twice. Upon the second write of 1 to the SOFTWARERESET bit, the DMA control is reset identically to a system reset. The logic which sequences the software reset process then resets itself automatically.

The data width of DMA transactions is specified by the BYTE, HW, WORD, DOUBLEWORD, and QUADWORD bits. Only one of these bits can be set at a time. If more than one of the bits is set, the DMA controller behavior is undefined. The width of the transfer is determined by the narrower of the two agents read and written. For example, a DMA transaction that reads from a 16-bit flash memory and writes to a 32-bit on-chip memory requires a halfword transfer. In this case, HW must be set to 1, and BYTE, WORD, DOUBLEWORD, and QUADWORD must be set to 0.

To successfully perform transactions of a specific width, that width must be enabled in hardware using the **Allowed Transaction** hardware option. For example, the DMA controller behavior is undefined if quadword transfers are disabled in hardware, but the QUADWORD bit is set during a DMA transaction.

Executing a DMA software reset when a DMA transfer is active may result in permanent bus lockup (until the next system reset). The SOFTWARERESET bit should therefore not be written except as a last resort.

29.4.4. Interrupt Behavior

The DMA controller has a single IRQ output that is asserted when the `status` register's DONE bit equals 1 and the control register's I_EN bit equals 1.

Writing the `status` register clears the DONE bit and acknowledges the IRQ. A host peripheral can read the `status` register and determine how the DMA transaction finished by checking the LEN, REOP, and WEOP bits.

29.5. DMA Controller Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
November 2017	2017.11.06	Clarified the values for REEN and WEEN bits in the *Table: Control Register Bits*.
December 2015	2015.12.12	Updated LEEN and WEEN in Control Register table.
June 2015	2015.06.12	Updated the GO bit description in the "Control Register Bits" table
July 2014	2014.07.24	Removed mention of SOPC Builder, updated to Platform Designer
December 2010	v10.1.0	Removed the "Device Support", "Instantiating the Core in SOPC Builder", and "Referenced Documents" sections.
July 2010	v10.0.0	Added a new parameter, **FIFO Depth**.
November 2009	v9.1.0	No change from previous release.
March 2009	v9.0.0	No change from previous release.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	v8.0.0	Updated the Functional Description of the core.

30. Modular Scatter-Gather DMA Core

30.1. Core Overview

In a processor subsystem, data transfers between two memory spaces can happen frequently. In order to offload the processor from moving data around a system, a Direct Memory Access (DMA) engine is introduced to perform this function instead. The Modular Scatter-Gather DMA (mSGDMA) is capable of performing data movement operations with preloaded instructions, called descriptors. Multiple descriptors with different transfer sizes, and source and destination addresses have the option to trigger interrupts.

The mSGDMA core has a modular design that facilitates easy integration with the FPGA fabric. The core consists of a dispatcher block with optional read host and write host blocks. The descriptor block receives and decodes the descriptor, and dispatches instructions to the read host and write host blocks for further operation. The block is also configured to transfer additional information to the host. In this context, the read host block reads data through its Avalon-MM host interface, and channels it into the Avalon-ST source interface, based on instruction given by the dispatcher block. Conversely, the write host block receives data from its Avalon-ST sink interface and writes it to the destination address via its Avalon-MM host interface.

30.2. Feature Description

The mSGDMA provides three configuration structures for handling data transfers between the Avalon-MM to Avalon-MM, Avalon-MM to Avalon-ST, and Avalon-ST to Avalon-MM modes. The sub-core of the mSGDMA is instantiated automatically according to the structure configured for the mSGDMA use model.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Figure 84. mSGDMA Module Configuration with support for Memory-Mapped Reads and Writes

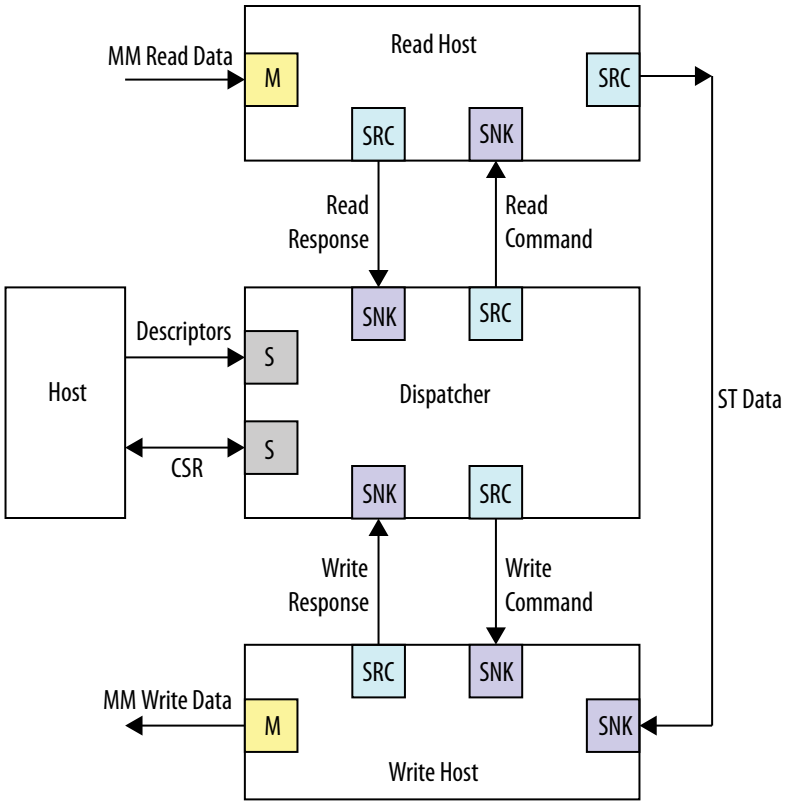


Figure 85. mSGDMA Module Configuration with Support for Memory-Mapped Streaming Reads to the Avalon-ST data bus

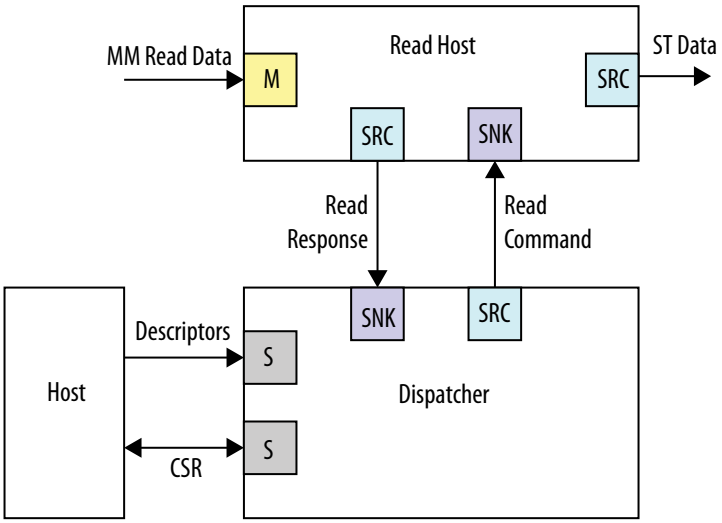
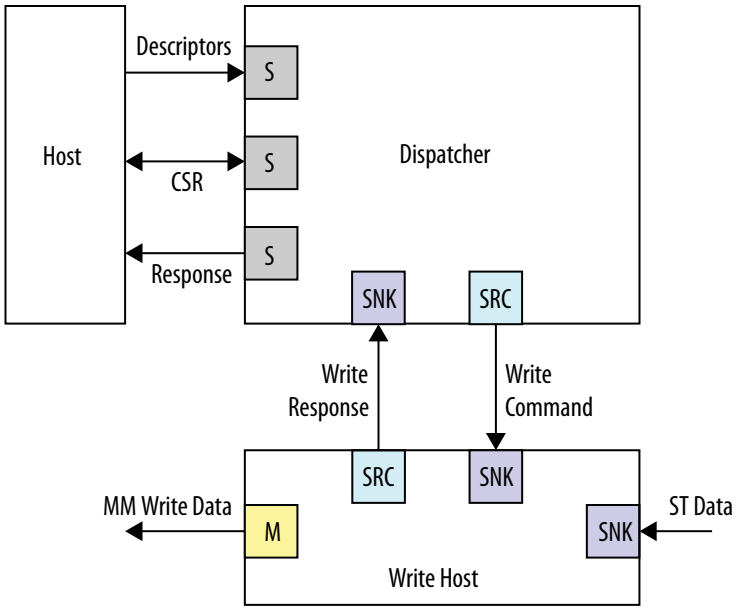


Figure 86. mSGDMA Module Configuration with Support for Avalon-ST Data Write Streaming to the Memory-Mapped Bus



The mSGDMA support 32-bit addressing by default. However, the core can support 64-bit addressing when you select **Extended Feature Options** in the parameter editor. It also supports extended features such as dynamic burst count programming, stride addressing, extended descriptor format (64-bit addressing), and unique sequence number identification for executed descriptor.

30.3. mSGDMA Interfaces and Parameters

30.3.1. Interface

The mSGDMA consists of the following:

- One Avalon-MM CSR agent port.
- One configurable Avalon-MM Agent or Avalon-ST Source Response port.
- Source and destination data path ports, which can be Avalon-MM or Avalon-ST.

The mSGDMA also provides an active-high-level interrupt output.

Only one clock domain can drive the mSGDMA. The requirement of different clock domains between source and destination data paths are handled by the Platform Designer fabric.

A hardware reset resets the whole system. Software reset resets the registers and FIFOs of the dispatchers of the dispatcher and host modules. For a software reset, read the resetting bit of the status register to determine when a full reset cycle completes.

30.3.1.1. Descriptor Agent Port

The descriptor agent port is write only and configurable to either 128 or 256 bits wide. The width is dependent on the descriptor format you choose for your system. When writing descriptors to this port, you must set the last bit high so the descriptor can be completely written to the dispatcher module. You can access the byte lanes of this port in any order, as long as the last bit is written to during the last write access.

30.3.1.2. Control and Status Register Agent Port

The control and status register (CSR) port is read/write accessible and is 32-bits wide. When the dispatcher response port is disabled or set to memory-mapped mode then the CSR port is responsible for sending interrupts to the host.

30.3.1.3. Response Port

The response port can be set to disabled, memory-mapped, or streaming. In memory-mapped mode the response information is communicated to the host via an Avalon-MM agent port. The response information is wider than the agent port, so the host must perform two read operations to retrieve all the information.

Note: Reading from the last byte of the response agent port performs a destructive read of the response buffer in the dispatcher module. As a result, always make sure that your software reads from the last response address last.

When you configure the response port to an Avalon Streaming source interface, connect it to a module capable of pre-fetching descriptors from memory. The following table shows the ST data bits and their description.

Table 285. Response Source Port Bit Fields

ST Data Bits	Description
31 - 0	Actual bytes transferred [31:0]
39 - 32	Error [7:0]
40	Early termination
41	Transfer complete IRQ mask
49 - 42	Error IRQ mask(26)
50	Early termination IRQ mask(26)
51	Descriptor buffer full(27)
255 - 52	Reserved

(26) Interrupt masks are buffered so that the descriptor pre-fetching block can assert the IRQ signal.

(27) Combinational signal to inform the descriptor pre-fetching block that space is available for another descriptor to be committed to the dispatcher descriptor FIFO(s).

30.3.1.4. Parameters

Table 286. Component Parameters

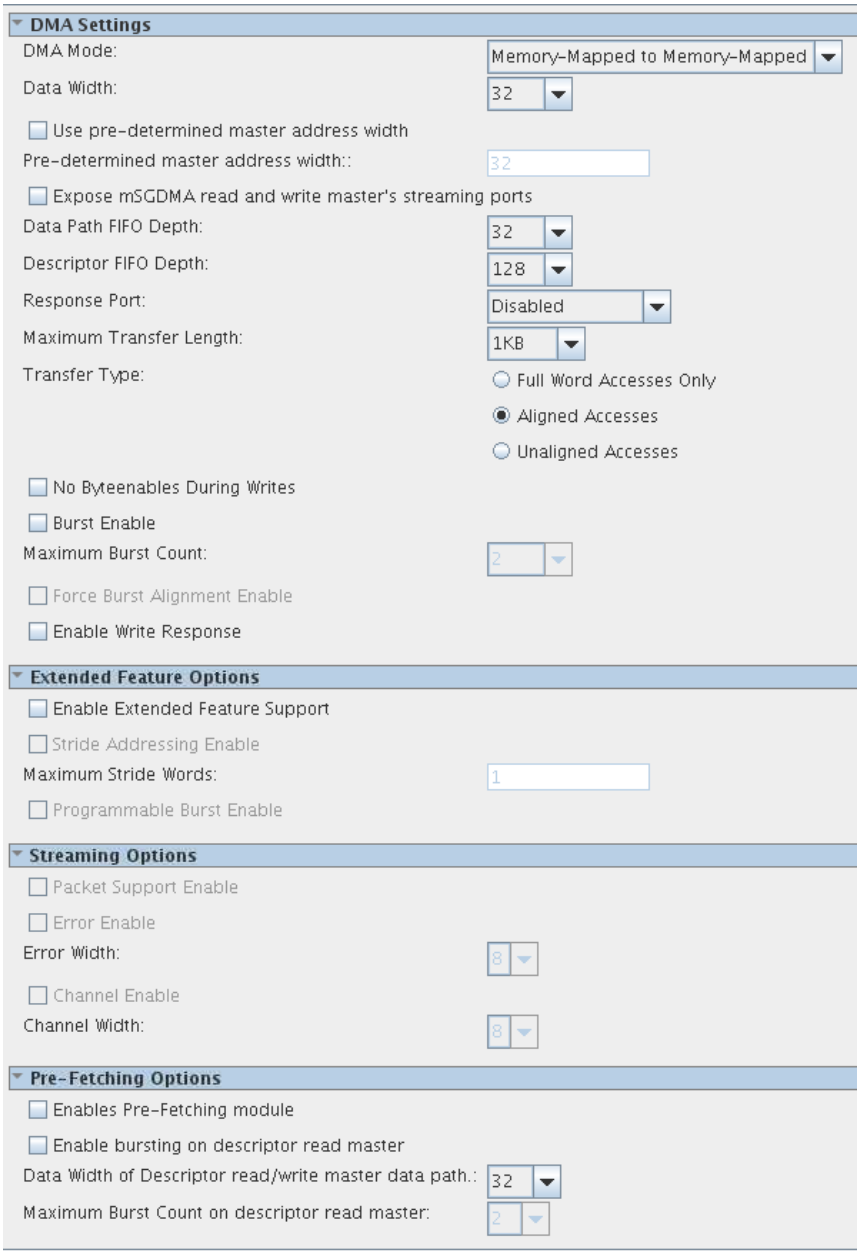
Parameter Name	Description	Allowable Range
DMA Mode	Transfer mode of mSGDMA. This parameter determines sub-cores instantiation to construct the mSGDMA structure.	Memory-Mapped to Memory-Mapped, Memory-Mapped to Streaming, Streaming to Memory-Mapped
Data Width	Data path width. This parameter affects both read host and write host data widths.	8, 16, 32, 64, 128, 256, 512, 1024
Use pre-determined host address width	Use pre-determined host address width instead of automatically-determined host address width.	Enable, Disable
Pre-determined host address width	Minimum host address width that is required to address memory agent.	32
Expose mSGDMA read and write host's streaming ports	When enabled, mSGDMA read host's data source port and mSGDMA write host's data sink port will be exposed for connection outside mSGDMA core.	Enable, Disable
Data Path FIFO Depth	Depth of internal data path FIFO.	16, 32, 64, 128, 256, 512, 1024, 2048, 4096
Descriptor FIFO Depth	FIFO size to store descriptor count.	8, 16, 32, 64, 128, 256, 512, 1024
Response Port	Option to enable response port and its port interface type	Memory-Mapped, Streaming, Disabled
Maximum Transfer Length	Maximum transfer length. With shorter length width being configured, the faster frequency of mSGDMA can operate in FPGA.	1KB, 2KB, 4KB, 8KB, 16KB, 32KB, 64KB, 128KB,256KB, 512KB, 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 128MB, 256MB, 512MB, 1GB, 2GB
Transfer Type	Supported transaction type	Full Word Accesses Only, Aligned Accesses, Unaligned Accesses
Burst Enable	Enable burst transfer	Enable, Disable
No Byteenables During Writes	When enabled, it forces all byte enables to high. This option is only applicable when transfer type is set to **Aligned Accesses** and DMA mode is set to either **Memory-mapped to Memory-Mapped** or **Streaming to Memory-Mapped**. When enabled, software need to handle scenario where transfer length is not multiple of data width at the end of transfer.	Enable, Disable
Maximum Burst Count	Maximum burst count	2, 4, 8, 16, 32, 64, 128, 256, 512, 1024
Force Burst Alignment Enable	Disable force burst alignment. Force burst alignment forces the hosts to post bursts of length 1 until the address is aligned to a burst boundary.	Enable, Disable
Enable Write Response	When enabled, it turns on the write response features of the write host. After completion of the DMA transfer, the host is only notified when all the outstanding writes have been responded.	Enable, Disable

continued...

Parameter Name	Description	Allowable Range
Enable Extended Feature Support	Enable extended features. In order to use stride addressing, programmable burst lengths, 64-bit addressing, or descriptor tagging the enhanced features support must be enabled.	Enable, Disable
Stride Addressing Enable	Enable stride addressing. Stride addressing allows the DMA to read or write data that is interleaved in memory. Stride addressing cannot be enabled if the burst transfer option is enabled.	Enable, Disable
Maximum Stride Words	Maximum stride amount (in words)	1 – 2G
Programmable Burst Enable	Enable dynamic burst programming	Enable, Disable
Packet Support Enable	Enable packetized transfer *Note:* When PACKET_ENABLE parameter is disabled and TRANSFER_TYPE is not "Full Word Accesses Only", any unaligned transfer length will cause additional bytes to be written during the last transfer beat of the Avalon streaming data source port of the read host core. Only with this parameter set TRUE, actual bytes transferred is meaningful for the transaction. PACKET_ENABLE only apply for ST-to-MM and MM-to-ST DMA operation mode.	Enable, Disable
Error Enable	Enable error field of ST interface	Enable, Disable
Error Width	Error field width	1, 2, 3, 4, 5, 6, 7, 8
Channel Enable	Enable channel field of ST interface	Enable, Disable
Channel Width	Channel field width	1, 2, 3, 4, 5, 6, 7, 8
Enable Pre-Fetching module	Enables prefetcher modules, a hardware core which fetches descriptors from memory.	Enable, Disable
Enable bursting on descriptor read host	Enable read burst will turn on the bursting capabilities of the prefetcher's read descriptor interface.	Enable, Disable
Data Width of Descriptor read/ write host data path	Width of the Avalon-MM descriptor read/write data path.	32, 64, 128, 256
Maximum Burst Count on descriptor read host	Maximum burst count.	Enable, Disable

30.3.2. mSGDMA Parameter Editor

Figure 87. Modular Scatter-Gather DMA Parameter Editor



30.4. mSGDMA Descriptors

The descriptor agent port is 128-bits for standard descriptors and 256-bits for extended descriptors. The tables below show acceptable standard and extended descriptor formats.

Table 287. Standard Descriptor Format

		Byte Lanes			
Offset	**Access**	**3**	**2**	**1**	**0**
0x0	Write	Read Address[31:0]			
0x4	Write	Write Address[31:0]			
0x8	Write	Length[31:0]			
0xC	Write	Control[31:0]			

Table 288. Extended Descriptor Format

		Byte Lanes			
Offset	**Access**	**3**	**2**	**1**	**0**
0x0	Write	Read Address[31:0]			
0x4	Write	Write Address[31:0]			
0x8	Write	Length[31:0]			
0xC	Write	Write Burst Count[7:0]	Read Burst Count [7:0]	Sequence Number[15:0]	
0x10	Write	Write Stride[15:0]		Read Stride[15:0]	
0x14	Write	Read Address[63:32]			
0x18	Write	Write Address[63:32]			
0x1C	Write	Control[31:0]			

All descriptor fields are aligned on byte boundaries and span multiple bytes when necessary. You can access each byte lane of the descriptor agent port independently of the others, allowing you to populate the descriptor using any access size.

Note: The location of the control field is dependent on the descriptor format you used. The last bit of the control field commits the descriptor to the dispatcher buffer when it is asserted. As a result, the control field is located at the end of a descriptor. This allows you to write the descriptor sequentially to the dispatcher block.

30.4.1. Read and Write Address Fields

The read and write address fields correspond to the source and destination address for each buffer transfer. Depending on the transfer type, you do not need to provide the read or write address. When performing memory-mapped to streaming transfers, the write address must not be written as there is no destination address. There is no destination address since the data is being transfered to a streaming port. Likewise, when performing streaming to memory-mapped transfers, the read address must not be written as the data source is a streaming port.

If a read or write address descriptor is written in a configuration that does not require it, the mSGDMA ignores the unnecessary address. If a standard descriptor is used and an attempt to write a 64-bit address is made, the upper 32 bits are lost and can cause the hardware to alias a lower address space. 64-bit addressing requires the use of the extended descriptor format.

30.4.2. Length Field

The length field is used to specify the number of bytes to transfer per descriptor. The length field is also used for streaming to memory-mapped packet transfers. This limits the number of bytes that can be transferred before the end-of-packet (EOP) arrives. As a result, you must always program the length field. If you do not wish to limit packet based transfers in the case of Avalon-ST to Avalon-MM transfers, program the length field with the largest possible value of 0xFFFFFFFF. This method allows you to specify a maximum packet size for each descriptor that has packet support enabled.

30.4.3. Sequence Number Field

The sequence number field is available only when using extended descriptors.The sequence number is an arbitrary value that you assign to a descriptor, so that you can determine which descriptor is being operated on by the read and write hosts. When performing memory-mapped to memory-mapped transfers, this value is tracked independently for hosts since each can be operating on a different descriptor. To use this functionality, program the descriptors with unique sequence numbers. Then, access the dispatcher CSR agent port to determine which descriptor is operated on.

30.4.4. Read and Write Burst Count Fields

The programmable read and write burst counts are only available when using the extended descriptor format. The programmable burst count is optional and can be disabled in the read and write hosts. Because the programmable burst count is an eight bit field for each host, the maximum that you can program burst counts of 1 to 128. Programming a value of zero or anything larger than 128 bits will be converted to the maximum burst count specified for each host automatically.

The maximum programmable burst count is 128 but when you instantiate the mSGDMA IP, you can have different selections up to 1024. Refer to the MAX_BURST_COUNT parameter in the parameter table. You will still have a burst count of 128 if you program for greater than 128. Programing to 0, gets the maximum burst count selected during instantiation time.

Related Information

Parameters on page 336
For more information, refer to the MAX_BURST_COUNT parameter.

30.4.5. Read and Write Stride Fields

The read and write stride fields are optional and only available when using the extended descriptor format. The stride value determines how the read and write hosts increment the address when accessing memory. The stride value is in terms of words, so the address incrementing is dependent on the host data width.

When stride is enabled, the host defaults to sequential accesses, which is the equivalent to a stride distance of one. A stride of zero instructs the host to continuously access the same address. A stride of two instructs the host to skip every other word in a sequential transfer. You can use this feature to perform interleaved data accesses, or to perform a frame buffer row and column transpose. The read and write stride distances are stored independently allowing, you to use different address incrementing for read and write accesses in memory-to-memory transfers. For example, to perform a 2:1 data decimation transfer, you would simply configure the

read host for a stride distance of two and the write host for a stride distance of one. To complete the decimation operation you could also insert a filter between the two hosts.

30.4.6. Control Field

The control field is available for both the standard and extended descriptor formats. This field can be programmed to configure parked descriptors, error handling, and interrupt masks. The interrupt masks are programmed into the descriptor so that interrupt enables are unique for each transfer.

Table 289. Descriptor Control Field Bit Definition

Bit	Sub-Field Name	Definition
31	Go	Commits all the descriptor information into the descriptor FIFO. As the host writes different fields in the descriptor, FIFO byte enables are asserted to transfer the write data to appropriate byte locations in the FIFO. However, the data written is not committed until the go bit has been written. As a result, ensure that the go bit is the last bit written for each descriptor. Writing '1' to the go bit commits the entire descriptor into the descriptor FIFO(s).
30:26	<reserved>	
25	Wait for write responses	When set, on completion of the DMA transfer, the host is only notified when all the outstanding writes have been responded. Those outstanding writes include writes transfer initiated by previous descriptor. This field is valid only when **Enable Write Response** parameter is set. Enabling this bit resulted in longer time for write response host to move into the next descriptor. Therefore, it is recommended to set this field on the last descriptor of the transfer.
24	Early done enable	Hides the latency between read descriptors. When the read host is set, it does not wait for pending reads to return before requesting another descriptor. Typically this bit is set for all descriptors except the last one. This bit is most effective for hiding high read latency. For example, it reads from SDRAM, PCIe, and SRIO.
23:16	Transmit Error / Error IRQ Enable	For for Avalon-MM to Avalon-ST transfers, this field is used to specify a transmit error. This field is commonly used for transmitting error information downstream to streaming components, such as an Ethernet MAC. In this mode, these control bits control the error bits on the streaming output of the read host. For Avalon-ST to Avalon-MM transfers, this field is used as an error interrupt mask. As errors arrive at the write host streaming sink port, they are held persistently. When the transfer completes, if any error bits were set at any time during the transfer and the error interrupt mask bits are set, then the host receives an interrupt. In this mode, these control bits are used as an error encountered interrupt enable.
15	Early Termination IRQ Enable	Signals an interrupt to the host when a Avalon-ST to Avalon-MM transfer completes early.

continued...

Bit	Sub-Field Name	Definition
		For example, if you set this bit and set the length field to 1MB for Avalon-ST to Avalon-MM transfers, this interrupt asserts when more than 1MB of data arrives to the write host without the end of packet being seen.
14	Transfer Complete IRQ Enable	Signals an interrupt to the host when a transfer completes. In the case of Avalon-MM to Avalon-ST transfers, this interrupt is based on the read host completing a transfer. In the case of Avalon-ST to Avalon-MM or Avalon-MM to Avalon-MM transfers, this interrupt is based on the write host completing a transfer.
13	<reserved>	
12	End on EOP	End on end of packet allows the write host to continuously transfer data during Avalon-ST to Avalon-MM transfers without knowing how much data is arriving ahead of time. This bit is commonly set for packet-based traffic such as Ethernet.
11	Park Writes	When set, the dispatcher continues to reissue the same descriptor to the write host when no other descriptors are buffered.
10	Park Reads	When set, the dispatcher continues to reissue the same descriptor to the read host when no other descriptors are buffered. This is commonly used for video frame buffering.
9	Generate EOP	Emits an end of packet on last beat of a Avalon-MM to Avlaon-ST transfer
8	Generate SOP	Emits a start of packet on the first beat of a Avalon-MM to Avalon-ST transfer
7:0	Transmit Channel	Emits a channel number during Avalon-MM to Avalon-ST transfers

30.5. Register Map of mSGDMA

The following table illustrates the mSGDMA register map under observation by host processor from its Avalon-MM CSR interfaces.

Table 290. CSR Registers Map

Byte Lanes					
Offset	**Attribute**	**3**	**2**	**1**	**0**
0x0	Read/Clear	Status			
0x4	Read/Write	Control			
0x8	Read	Write Fill Level[15:0]		Read Fill Level[15:0]	
0xC	Read	<reserved>[28]		Response Fill Level[15:0]	
0x10	Read	Write Sequence Number[15:0][29]		Read Sequence Number[15:0][29]	

continued...

(28) Writing to reserved bits will have no impact on the hardware, reading will return unknown data.

(29) Sequence numbers will only be present when dispatcher enhanced features are enabled.

Byte Lanes					
Offset	Attribute	3	2	1	0
0x14	Read	Component Configuration 1[30]			
0x18	Read	Component Configuration 2[30]			
0x1C	Read	<reserved>[28]		Component Type [7:0][31]	Component Version [7:0][32]

30.5.1. Status Register

Table 291. Status Register Bit Definition

Bit	Name	Description
31:10	<reserved>	N/A
9	IRQ	Set when interrupt condition occurs. This bit is set by hardware and cleared by software. To clear this bit, software needs to write a 1 to this bit. This bit is set when a hardware event has a higher priority than a clear by a software event.
8	Stopped on Early Termination	When the dispatcher is programmed to stop on early termination, this bit is set. Also set, when the write host is performing a packet transfer and does not receive EOP before the pre-determined amount of bytes are transferred, which is set in the descriptor length field. If you do not wish to use early termination you should set the transfer length of the descriptor to 0xFFFFFFFF ,which gives you the maximum packet based transfer possible (early termination is always enabled for packet transfers).
7	Stopped on Error	When the dispatcher is programmed to stop errors and when an error beat enters the write host the bit is set.
6	Resetting	Set when you write to the software reset register and the SGDMA is in the middle of a reset cycle. This reset cycle is necessary to make sure there are no incoming transfers on the fabric. When this bit de-asserts you may start using the SGDMA again.
5	Stopped	Set when you either manually stop the SGDMA, or you setup the dispatcher to stop on errors or early termination and one of those conditions occurred. If you manually stop the SGDMA this bit is asserted after the host completes any read or write operations that were already in progress.
4	Response Buffer Full	Set when the response buffer is full.
3	Response Buffer Empty	Set when the response buffer is empty.
2	Descriptor Buffer Full	Set when either the read or write command buffers are full.
1	Descriptor Buffer Empty	Set when both the read and write command buffers are empty.
0	Busy	Set when

continued...

(30) Only available in Intel Quartus Prime Pro Edition. In Intel Quartus Prime Standard Edition, these fields are reserved.

(31) Fix value: 0xDA. Only available in Intel Quartus Prime Pro Edition. In Intel Quartus Prime Standard Edition, these fields are reserved.

(32) Fix value: 0x1. Only available in Intel Quartus Prime Pro Edition. In Intel Quartus Prime Standard Edition, these fields are reserved.

Bit	Name	Description
		• the dispatcher still has commands buffered • one of the hosts is still transferring data • the write host is waiting for write responses to return if wait for write response is enabled in the descriptor and write responses are still returning.

30.5.2. Control Register

Table 292. Control Register Bit Definition

Bit	Name	Description
31:6	<reserved>	Reserved.
5	Stop Descriptors	Setting this bit stops the SGDMA dispatcher from issuing more descriptors to the read or write hosts. Read the stopped status register to determine when the dispatcher stopped issuing commands and the read and write hosts are idle.
4	Global Interrupt Enable Mask	Setting this bit will allow interrupts to propagate to the interrupt sender port. This mask occurs before the register logic so any interrupt events that are triggered when the mask is disabled will not be latched by IRQ register bit in status register.
3	Stop on Early Termination	Setting this bit stops the SGDMA from issuing more read/write commands to the host modules if the write host attempts to write more data than the user specifies in the length field for packet transactions. The length field is used to limit how much data can be sent and is always enabled for packet based writes.
2	Stop on Error	Setting this bit stops the SGDMA from issuing more read/write commands to the host modules if an error enters the write host module sink port.
1	Reset Dispatcher	Setting this bit resets the registers and FIFOs of the dispatcher and host modules. Since resets can take multiple clock cycles to complete due to transfers being in flight on the fabric, you should read the resetting status register to determine when a full reset cycle has completed.
0	Stop Dispatcher	Setting this bit stops the SGDMA in the middle of a transaction. If a read or write operation is occurring, then the access is allowed to complete. Read the stopped status register to determine when the SGDMA has stopped. After reset, the dispatcher core defaults to a start mode of operation.

The response agent port of mSGDMA contains registers providing information of the executed transaction. This register map is only applicable when the response mode is enabled and set to memory mapped. Also when the response port is enabled, it needs to have responses read because it buffers responses. When setup as a memory-mapped agent port, reading byte offset 0x7 outputs the response. If the response FIFO becomes full the dispatcher stops issuing transfer commands to the read and write hosts. The following describes the registers definition.

Table 293. Response Registers Map

Byte Lanes					
Offset	**Access**	**3**	**2**	**1**	**0**
0x0	Read	Actual Bytes Transferred[31:0]			
0x4	Read	<reserved>(33)	<reserved>	Early Termination(34)	Error[7:0]

(33) Reading from byte 7 outputs the response FIFO.

The following list explains each of the fields:

- **Actual bytes transferred** determines how many bytes transferred when the mSGDMA is configured in Avalon-ST to Avlaon-MM mode with packet support enabled. Since packet transfers are terminated by the IP providing the data, this field counts the number of bytes between the start-of-packet (SOP) and end-of-packet (EOP) received by the write host. If the early termination bit of the response is set, then the actual bytes transferred is an underestimate if the transfer is unaligned.
- **Error** Determines if any errors were issued when the mSGDMA is configured in Avalon-ST to Avlaon-MM mode with error support enabled. Each error bit is persistent so that errors can accumulate throughout the transfer.
- **Early Termination** determines if a transfer terminates because the transfer length is exceeded when the SGDMA is configured in Avalon-ST to Avalon-MM mode with packet support enabled. This bit is set when the number of bytes transferred exceeds the transfer length set in the descriptor before the end-of-packet is received by the write host.

30.5.3. Write Fill Level Register

Bit	Name	Description
15:0	Write Fill Level	Indicates number of valid descriptor entries in the write command fifo that are yet to be executed. The fill level read back of 0 indicates that the write host is either idle or operating on the last descriptor.

30.5.4. Read Fill Level Register

Bit	Name	Description
15:0	Read Fill Level	Indicates number of valid descriptor entries in the read command fifo that are yet to be executed. The fill level read back of 0 indicates that the read host is either idle or operating on the last descriptor.

30.5.5. Response Fill Level Register

Bit	Name	Description
15:0	Response Fill Level	Indicates number of valid response entries in the response fifo that are yet to be retrieved. Response data can be retrieved from response fifo through either response Streaming port or Memory-Mapped agent port. If the **Response Port** parameter is disabled, this field output is 0.

30.5.6. Write Sequence Number Register

Bit	Name	Description
15:0	Write Sequence Number	Indicates the sequence number of the next descriptor for the write host to operate on. If transfer mode is set to Memory-Mapped to Streaming, the output value is 0. This field is available only when **Extended Feature Support** parameter is enabled.

(34) Early Termination is a single bit located at bit 8 of offset 0x4.

30.5.7. Read Sequence Number Register

Bit	Name	Description
15:0	Read Sequence Number	Indicates the sequence number of the next descriptor for the read host to operate on. If transfer mode is set to Memory-Mapped to Streaming, the output value is 0. This field is available only when **Extended Feature Support** parameter is enabled.

30.5.8. Component Configuration 1 Register

Bit	Name	Description
0	BURST_ENABLE	Indicates burst transfer: • 0: Burst mode is disabled • 1: Burst mode is enabled
1	BURST_WRAPPING_SUPPORT	Indicates burst wrapping support: • 0: Burst wrapping is disabled • 1: Burst wrapping is enabled
2	CHANNEL_ENABLE	Indicates channel support in data streaming interface: • 0: Channel support is disabled • 1: Channel support is enabled
5:3	CHANNEL_WIDTH	Indicates the number of channel used in data streaming interface. The number of channel is CHANNEL_WIDTH + 1: • 0: 1 channel • 1: 2 channels • 2: 3 channels • ... • 7: 8 channels
9:6	DATA_FIFO_DEPTH	Indicates the depth of internal data path fifo. The depth of the data path fifo is $2^{(DATA_FIFO_DEPTH + 4)}$: • 0: Depth of 16 • 1: Depth of 32 • 2: Depth of 64 • ... • 8: Depth of 4096 • 9 to 15 : Reserved
12:10	DATA_WIDTH	Indicates the width of data path. The width of data path is $2^{(DATA_WIDTH + 3)}$: • 0: Width of 8 • 1: Width of 16 • 2: Width of 32 • ... • 7: Width of 1024
15:13	DESCRIPTOR_FIFO_DEPTH	Indicates the depth of descriptor fifo. The depth of descriptor fifo is $2^{(DESCRIPTOR_FIFO_DEPTH + 3)}$: • 0: Width of 8 • 1: Width of 16 • 2: Width of 32 • ... • 7: Width of 1024
17:16	DMA_MODE	Indicates the transfer mode:

continued...

Bit	Name	Description
		• 0: Memory-Mapped to Memory-Mapped • 1: Memory-Mapped to Streaming • 2: Streaming to Memory-Mapped • 3: Reserved
18	ENHANCED_FEATURES	Indicates extended features support: • 0: Extended features is disabled • 1: Extended features is enabled
19	ERROR_ENABLE	Indicates error support in data streaming interface: • 0: Error support is disabled • 1: Error support is enabled
22:20	ERROR_WIDTH	Indicates the number of error lines in data streaming interface. The number of error lines is ERROR_WIDTH + 1: • 0: 1 error line • 1: 2 error lines • 2: 3 error lines • ... • 7: 8 error lines
26:23	MAX_BURST_COUNT	Indicates the maximum burst count. The maximum burst count is $2^{(MAX_BURST_COUNT + 1)}$: • 0: Burst count of 2 • 1: Burst count of 4 • 2: Burst count of 8 • ... • 9: Burst count of 1024 • 10 to 15 : Reserved
31:27	MAX_BYTE	Indicates the maximum transfer length. The maximum transfer length is $2^{(MAX_BYTE + 10)}$: • 0: Transfer length of 1024 • 1: Transfer length of 2048 • 2: Transfer length of 4096 • ... • 21: Transfer length of 2147483648

30.5.9. Component Configuration 2 Register

Bit	Name	Description
0	STRIDE_ENABLE	Indicates stride addressing: • 0: Stride addressing is disabled • 1: Stride addressing is enabled
15:1	MAX_STRIDE	Indicates maximum stride count. The actual stride count is MAX_STRIDE + 1
16	PACKET_ENABLE	Indicates packet transfer: • 0: Packet transfer is disabled • 1: Packet transfer is enabled
17	PREFETCHER_ENABLE	Indicates pre-fetching mode: • 0: Pre-fetching mode is disabled • 1: Pre-fetching mode is enabled
18	PROGRAMMABLE_BURST_ENABLE	Indicates dynamic burst programming:

continued...

Bit	Name	Description
		• 0: Dynamic burst programming is disabled • 1: Dynamic burst programming is enabled
20:19	RESPONSE_PORT	Indicates response port and its port interface type: • 0: Memory-Mapped • 1: Streaming • 2: Disabled • 3: Reserved
22:21	TRANSFER_TYPE	Indicates transaction type: • 0: Full word accesses only • 1: Aligned accesses • 2: Unaligned accesses • 3: Reserved
31:23	-	Reserved

30.5.10. Component Type Register

Bit	Name	Description
7:0	Type	Specifies component type number. Fix value: 0xDA. Only available in Intel Quartus Prime Pro Edition. In Intel Quartus Prime Standard Edition, these fields are reserved.

30.5.11. Component Version Register

Bit	Name	Description
7:0	Version	Specifies component version number. Fix value: 0x1. Only available in Intel Quartus Prime Pro Edition. In Intel Quartus Prime Standard Edition, these fields are reserved.

30.6. Programming Model

30.6.1. Stop DMA Operation

The stop DMA operation is also referring to stop dispatcher. Once the "Stop Dispatcher" bit is set in the Control Register, no further new read or write transaction is issued. However, existing transactions pending completion are allowed to complete. The command buffer in both the read host and write host must be clear before the DMA resumes operation via a reset request. Proceed with the following steps for the stop DMA operation:

1. Set the **Stop Dispatcher** bit of the Control Register.
2. Recursively check if **Stopped** bit of Status Register is asserted.
3. When the **Stopped** bit of the Status Register is asserted, reset the DMA by setting the **Reset Dispatcher** bit of the Control Register. When setting the **Reset Dispatcher** bit of Control Register, ensures **Stop DMA** bit remains in set condition.
4. Check if the **Resetting** bit of the Status Register is deasserted. If it is, DMA is now back in normal operation.

30.6.2. Stop Descriptor Operation

The Stop Descriptor temporary stops the dispatcher core from continuing to issue commands to the read host and write host. The dispatcher core operates in the sense that it can accept a descriptor sent by the host up to its descriptor FIFO limit. Proceed with the following steps for the stop descriptor operation:

1. Set "Stop Descriptor" bit of Control Register.
2. Check if "Stopped" bit of Status Register is asserted.

To resume DMA from its previously stop descriptor operation, do the following:

1. Unset the "Stop Descriptor" bit of Control Register.
2. Check if "Stopped" bit of Status Register is deasserted.

30.6.3. Recovery from Stopped on Error and Stopped on Early Termination

When stopped on error or stopped on early termination occurs, mSGDMA requires a software reset to continue operation.

1. When the "Stopped" bit of the Status register is asserted, reset the DMA by setting the "Reset Dispatcher" bit of Control register.
2. Check if the "Resetting" bit of Status register is deasserted. If it is, DMA is now back in normal operation.

30.7. Modular Scatter-Gather DMA Prefetcher Core

The mSGDMA Prefetcher core is an additional micro core to the existing mSGDMA core. The Prefetcher core provides extra functionality through the Avalon-MM and dispatcher core. The Avalon-MM fetches a series of descriptors from memory that describes the required data transfers before passing them to dispatcher core for data transfer execution. The series of descriptors in memory can be linked together to form a descriptor list. This allows the DMA core to execute multiple descriptors in single run, thus enabling transfer to a non-contiguous memory space and improves system performance.

30.7.1. Functional Description

30.7.1.1. Supported Features

- Descriptor linked list
- Data transfer to non-contiguous memory space
- Descriptor write back
- Hardware descriptor polling
- 64-bit address spaces

30.7.1.2. Architecture Overview

The Prefetcher core supports all the three existing Modular SGDMA configurations:

- Memory-Mapped to Memory-Mapped
- Memory-Mapped to Streaming
- Streaming to Memory-Mapped

On interfaces facing host and external peripherals, it has dedicated Avalon-MM read and write host interfaces to fetch series of descriptors from memory as well as performing a descriptor write back. It has one Avalon Memory-Mapped CSR agent interface for the host processor to access the configuration register in the Prefetcher core.

On interfaces facing the internal dispatcher core, it has an Avalon-MM descriptor write host interface to write a descriptor to the dispatcher core. It has Avalon-ST response sink interface to receive response information from the dispatcher core upon completion of each descriptor execution.

Figure 88. Memory-Mapped to Memory-Mapped Configuration with Prefetcher Enabled

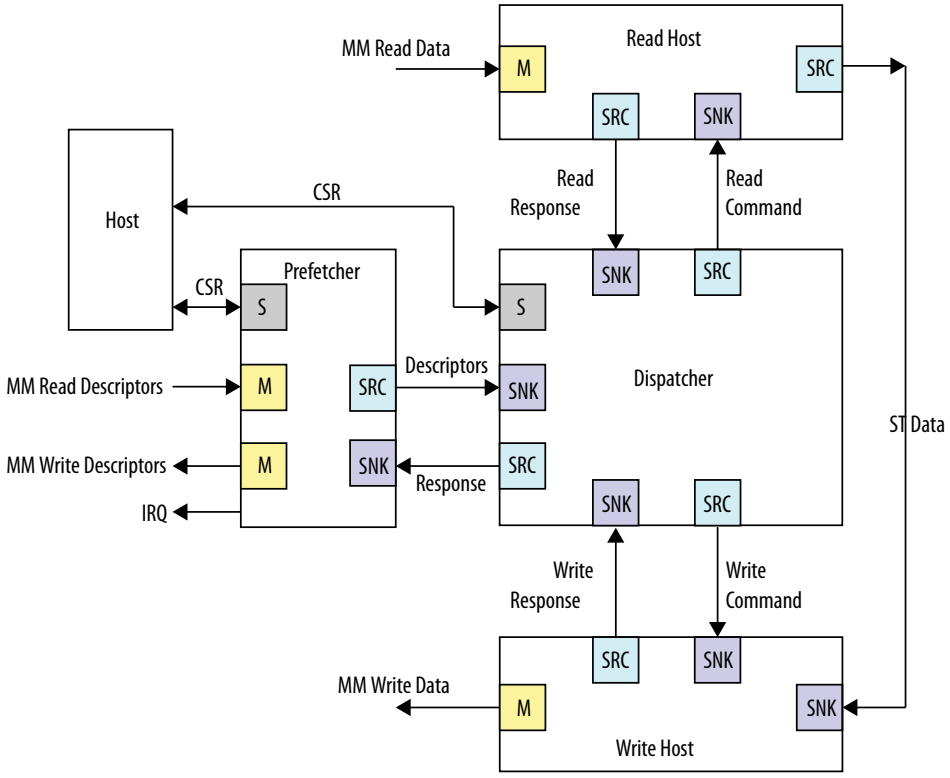


Figure 89. Memory-Mapped to Streaming Configuration with Prefetcher Enabled

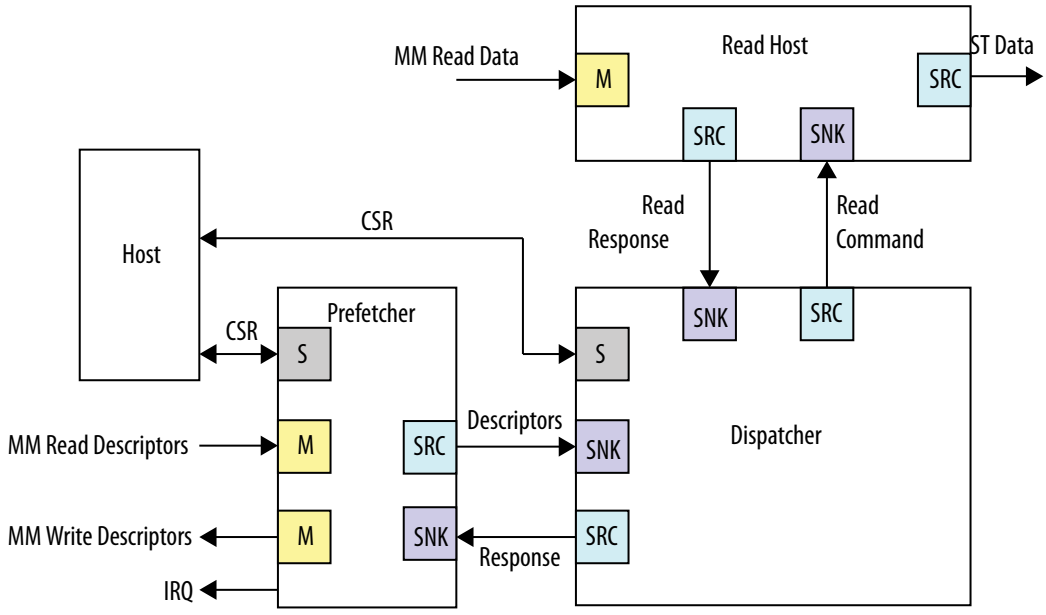
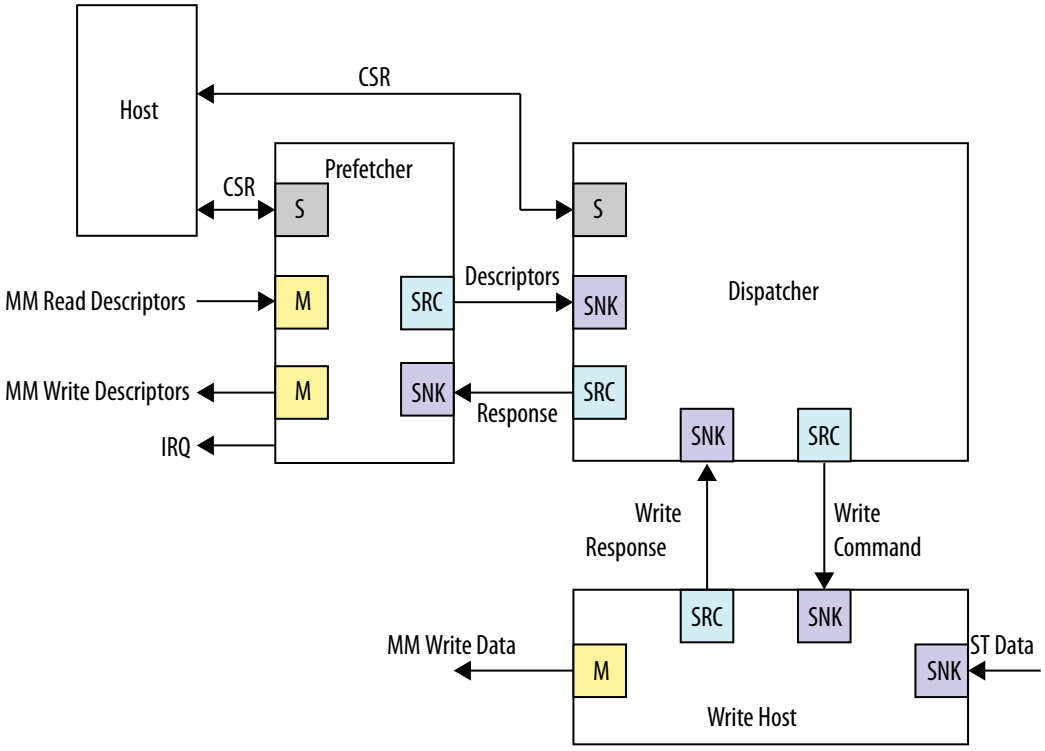


Figure 90. Streaming to Memory-Mapped Configuration with Prefetcher Enabled



30.7.1.3. Descriptor Format

The mSGDMA without the Prefetcher core defines two types of descriptor formats. Standard descriptor format which consists of 128 bits and extended descriptor format which consists of 256 bits. With the Prefetcher core enabled, the existing descriptor format is expanded to 256 bits and 512 bits respectively in order to accommodate additional control information for the prefetcher operation.

Table 294. Standard Descriptor Format when Prefetcher is Enabled

	Byte Lanes			
Offset	3	2	1	0
0x0	Read Address [31-0]			
0x4	Write Address [31-0]			
0x8	Length [31-0]			
0xC	Next Desc Ptr [31-0]			
0x10	Actual Bytes Transferred [31-0]			
0x14	Reserved [15-0]		Status [15-0]	
0x18	Reserved [31-0]			
0x1C	Control [31, 30, 29..0]			

Table 295. Extended Descriptor Format when Prefetcher is Enabled

	Byte Lanes			
Offset	3	2	1	0
0x0	Read Address [31-0]			
0x4	Write Address [31-0]			
0x8	Length [31-0]			
0xC	Next Desc Ptr [31-0]			
0x10	Actual Bytes Transferred [31-0]			
0x14	Reserved [15-0]		Status [15-0]	
0x18	Reserved [31-0]			
0x1C	Write Burst Count [7-0]	Read Burst Count [7-0]	Sequence Number [15-0]	
0x20	Write Stride [15-0]		Read Stride [15-0]	
0x24	Read Address [63-32]			
0x28	Write Address [63-32]			
0x2C	Next Desc Ptr [63-32]			
0x30	Reserved [31-0]			
0x34	Reserved [31-0]			
0x38	Reserved [31-0]			
0x3C	Control [31, 30, 29..0]			

30.7.1.3.1. Descriptor Fields Definition

Next Descriptor Pointer

The next descriptor pointer field specifies the address of the next descriptor in the linked list.

Actual Bytes Transferred

Specifies the actual number of bytes that has been transferred. This field is not applicable if Modular SGDMA is configured as Memory-Mapped to Streaming transfer.

Table 296. Status

Bits	Fields	Description
15:9	Reserved	Reserved fields
8	Early Termination	Indicates early termination condition where write host is performing a packet transfer and does not receive EOP before pre-determined amount of bytes are transferred. This status bit is similar to status register bit 8 of the dispatcher core. For more details refer to dispatcher core CSR definition. This field is not applicable if Modular SGDMA is configured as Memory-Mapped to Streaming transfer.
7:0	Error	Indicates an error has arrived at the write host streaming sink port. This field is not applicable if Modular SGDMA is configured as Memory-Mapped to Streaming transfer.

Table 297. Control

Bits	Fields	Description
30	Owned by Hardware	This field determines whether hardware or software has write access to the current descriptor. When this field is set to 1, the Modular SGDMA can update the descriptor and software should not access the descriptor due to the possibility of race conditions. Otherwise, it is safe for software to update the descriptor.

For bit 31 and 29:0, refer to descriptor control field bit 31 and 29:0 defined in dispatcher core. Table 289 on page 341

30.7.1.3.2. Descriptor Processing

The DMA descriptors specify data transfers to be performed. With the Prefetcher core, a descriptor is stored in memory and accessed by the Prefetcher core through its descriptor write and descriptor read Avalon-MM host. The mSGDMA has an internal FIFO to store descriptors read from memory. This FIFO is located in the dispatcher's core. The descriptors must be initialized and aligned on a descriptor read/write data width boundary. The Prefetcher core relies on a cleared Owned By Hardware bit to stop processing. When the Owedn by Hardware bit is 1, the Prefetcher core goes ahead to process the descriptor. When the Owned by Hardware bit is 0, the Prefetcher core does not process the current descriptor and assumes the linked list has ended or the next descriptor linked list is not yet ready.

Each time a descriptor has been processed, the core updates the Actual Byte Transferred, Status and Control fields of the descriptor in memory (descriptor write back). The Owned by Hardware bit in the descriptor control field is cleared by the core during descriptor write back. Refer to software programming model section to know more about recommended way to set up the Prefetcher core, building and updating the descriptor list.

In order for the Prefetcher to know which memory addresses to perform descriptor write back, the next descriptor pointer information will need to be buffered in Prefetcher core. This buffer depth will be similar to descriptor FIFO depth in dispatcher core. This information is taken out from buffer each time a response is received from dispatcher.

30.7.1.4. Registers

30.7.1.4.1. Register Map

Table 298. Register Map

Name	Address Offset	Description
Control	0x0	Specifies the Prefetcher core behavior such as when to start the core.
Next Descriptor Pointer Low	0x1	Contains the address [31:0] of the next descriptor to process. Software sets this register to the address of the first descriptor as part of the system initialization sequence. If descriptor polling is enabled, this register is also updated by hardware to store the latest next descriptor

continued...

Name	Address Offset	Description
		address. The latest next descriptor address is used by the Prefetcher core to perform descriptor polling.
Next Descriptor Pointer High	0x2	Contains the address [63:32] of the next descriptor to process. Software set this register to the address of the first descriptor as part of the system initialization sequence. This field is used only when higher than 32-bit addressing is used when mSGDMA's extended feature is enabled. If descriptor polling is enabled, this register is also updated by hardware to store the latest next descriptor address. The latest next descriptor address is used by the Prefetcher core to perform descriptor polling.
Descriptor Polling Frequency	0x3	Descriptor Polling Frequency
Status	0x4	Status Register

30.7.1.4.2. Control Register

The address offset for the Control Register table is 0x0.

Table 299. Control Register

Bit	Fields	Access	Default Value	Description
31:5	Reserved	R	0x0	Reserved fields
4	Park Mode	R/W	0x0	This bit enables the mSGDMA to repeatedly execute the same linked list over and over again. In order for this to work, software need to setup the last descriptor to point back to the first descriptor. 1: Park mode is enabled. Pefetcher will not clear the owned by hardware field during descriptor write back 0: Park mode is disabled. Prefetcher will clear the owned by hardware field during descriptor write back. Software can terminate the park mode operation by clearing this field. Since this field is in CSR and not in descriptor field itself, this termination event is asynchronous to current descriptor in progress (user can't deterministically choose which descriptor in the linked list to stop). Park mode feature is not intended to be used on the fly. User must not enable this bit when the Prefetcher is already in operation. This bit shall be set during initialization/configuration phase of the control register.
3	Global Interrupt Enable Mask	R/W	0x0	Setting this bit will allow interrupts to propagate to the interrupt sender port. This mask occurs before the register logic so any interrupt events that are triggered

continued...

Bit	Fields	Access	Default Value	Description
				when the mask is disabled will not be latched by IRQ register bit in status register. *Note:* There is an equivalent global interrupt enable mask bit in dispatcher core CSR. When the Prefetcher is enabled, software shall use this bit. When the Prefetcher is disabled, software shall use equivalent global interrupt enable mask bit in dispatcher core CSR.
2	Reset_Prefetcher	R/W1S[35]	0x0	This bit is used when software intends to stop the Prefetcher core when it is in the middle of data transfer. When this bit is 1, the Prefetcher core begin its reset sequence. This bit is automatically cleared by hardware when the reset sequence has completed. Therefore, software need to poll for this bit to be cleared by hardware to ensure the reset sequence has finished. This function is intended to be used along with reset dispatcher function in dispatcher core. Once the reset sequence in the Prefetcher core has completed, software is expected to reset the dispatcher core, polls for dispatcher's reset sequence to be completed by reading dispatcher core status register.
1	Desc_Poll_En	R/W	0x0	Descriptor polling enable bit. 1: When the last descriptor in current linked list has been processed, the Prefetcher core polls the Owned By Hardware bit of next descriptor to be set and automatically resumes data transfer without the need for software to set the Run bit. The polling frequency is specified in Desc_Poll_Freq register. 0: When the last descriptor in current linked list has been processed, the Prefetcher stops operation and clears the run bit. In order to restart the DMA engine, software need to set the Run bit back to 1. In case software intends to disable polling operation in the middle of transfer, software can write this field to 0. In this case, the whole mSGDMA operation is stopped when the Prefetcher core encounter owned by hardware bit = 0. *Note:* This bit should be set during initialization or configuration of the control register.
0	Run	R/W1S	0x0	Software sets this bit to 1 to start the descriptor fetching operation which subsequently initiates the DMA transaction.

(35) W1S register attribute means, software can write 1 to set the field. Software write 0 to this field has no effect.

Bit	Fields	Access	Default Value	Description
				When descriptor polling is disabled, this bit is automatically cleared by hardware when the last descriptor in the descriptor list has been processed or when the Prefetcher core read owned by hardware bit = 0. When descriptor polling is enabled, mSGDMA operation is continuously run. Thus the run bit stays 1. This field is also cleared by hardware when reset sequence process triggered by Reset_Prefetcher bit completes.

30.7.1.4.3. Descriptor Polling Frequency

Table 300. Desc_Poll_Freq

Bit	Fields	Access	Default	Description
31:16	Reserved	R	0x0	Reserved fields
15:0	Poll_Freq	R/W	0x0	Specifies the frequency of descriptor polling operation. The polling frequency is in term of number of clock cycles. The poll period is counted from the point where read data is received by the Prefetcher core.

30.7.1.4.4. Status

Table 301. Status

Bit	Fields	Access	Default Value	Description
31:1	Reserved	R	0x0	Reserved fields
0	IRQ	R/W1C[36]	0x0	Set by hardware when an interrupt condition occurs. Software must perform a write 1 to this field in order to clear it. There is an equivalent IRQ status bit in the dispatcher core CSR. When the Prefetcher is enabled, software uses this bit as an IRQ status indication. When the Prefetcher is disabled, software uses equivalent IRQ status bit in dispatcher core CSR.

(36) W1C register attribute means, software write 1 to clear the field. Software write 0 to this field has no effect.

30.7.1.5. Interfaces

30.7.1.5.1. Avalon-MM Read Descriptor

This interface is used to fetch descriptors in memory. It supports non-burst or burst mode which configurable during generation time.

Table 302. Avalon-MM Read Descriptor

Signal Role	Width	Description
Address	32 to 64-bit	Avalon-MM read address. 32-bits if extended feture is disabled. 32- to 64-bits if extended feature is enabled.
Read	1	Avalon-MM read control
Read data	32, 64, 128, 256, 512	Avalon-MM read data bus. Data width is configurable during IP generation.
Wait request	1	Avalon-MM wait request for backpressure control.
Read data valid	1	Avalon-MM read data valid indication.
Burstcount	1/2/3/4/5	Avalon-MM burst count. The maximum burst count is configurable during IP generation. This signal role is applicable only when the Enable Bursting on the descriptor read host is turned on.

30.7.1.5.2. Avalon-MM Write Descriptor

This interface is used to access the Prefetcher CSR registers. It has fixed write and read wait time of 0 cycles and read latency of 1 cycle.

Table 303. Avalon-MM Write Descriptor

Signal Role	Width	Description
Address	32 to 64	Avalon-MM write address
Write	1	Avalon-MM read control
Wait request	1	Avalon-MM waitrequest for backpressure control
Write data	32, 64, 128, 256, 512	Avalon-MM write data bus
Byte enable	4, 8, 16, 32, 64	Avalon-MM write byte enable control. Its width is automatically derived from selected data width

30.7.1.5.3. Avalon-MM CSR

This interface is used to access the Prefetcher CSR registers. It has fixed write and read wait time of 0 cycles and read latency of 1 cycle.

Table 304. Avalon-MM CSR

Signal Role	Width	Description
Address	3	Avalon-MM write address
Write	1	Avalon-MM read control
Read	1	Avalon-MM write control
Write data	32	Avalon-MM write data bus
Read data	32	Avalon-MM read data bus

30.7.1.5.4. Avalon-ST Descriptor Source

This interface is used by the Prefetcher to write descriptor information into the dispatcher core.

Table 305. Avalon-ST Descriptor Source

Signal Role	Width	Description
Valid	1	Avalon-ST valid control
Ready	1	Avalon-ST ready control with ready latency of 0. Refer to dispatcher's descriptor format for write data definition.
Data	128/256	Avalon-ST data bus

30.7.1.5.5. Avalon-ST Response

This interface is used by the Prefetcher core to retrieve response information from dispatcher’s core upon each transfer completion.

Table 306. Avalon-ST Response

Signal Role	Width	Description
Valid	1	Avalon-ST valid control. Prefetcher core expects valid signal to remain high while the bus is being back pressured.
Ready	1	Avalon-ST ready control. Used by the Prefetcher core to back pressure the external ST response source.
Data	256	Avalon-ST data bus. Refer to dispatcher’s response source format for ST data definition. Prefetcher core expects data signals to remain constant while the bus is being back pressured.

Streaming interface (ST) data bus format and definition are similar to the dispatcher's response source format:

Table 307. Avalon-ST Response Data Format and Definition

Bits	Signal Information
[31:0]	Actual bytes transferred [31:0]
[39:32]	Error [7:0]
40	Early termination
41	Transfer complete IRQ mask
[49:42]	Error IRQ mask
50	Early termination IRQ mask
51	Descriptor buffer full
[255:52]	Reserved

30.7.1.5.6. IRQ Interface

When the Prefetcher is enabled, IRQ generation no longer outputs from the dispatcher's core. It will be outputted from the Prefetcher core. The sources of the interrupt remain the same which are transfer completion, early termination, and error detection. Masking bits for each of the interrupt sources are programmed in the descriptor. This information will be passed to the Prefetcher core through the ST response interface. An equivalent global interrupt enable mask and IRQ status bit which are defined in dispatcher core are now defined in the Prefetcher core as well. These two bits need to be defined in the Prefetcher core since the actual IRQ register is now located in the Prefetcher core.

30.7.1.6. Software Programming Model

30.7.1.6.1. Setting up Descriptor and mSGDMA Configuration Flow

The following is the recommended software flow to setup the descriptor and configuring the mSGDMA.

1. Build the descriptor list and terminate the list with a non-hardware owned descriptor (Owned By Hardware = 0).
2. Configure mSGDMA by accessing dispatcher core control register (for example: to configure Stop on Error, Stop on Early Termination, etc…)
3. Configure mSGDMA by accessing the Prefetcher core configuration register (for example: to write the address of the first descriptor in the first list to the next descriptor pointer register and set the Run bit to 1 to initiate transfers).
4. While the core is processing the first list, your software may build a second list of descriptors.

5. An IRQ can be generated each time a descriptor transfer is completed (depends whether transfer complete IRQ mask is set for that particular descriptor). If you only need an IRQ to be generated when mSGDMA finishes processing the first list, you only need to set transfer complete IRQ mask for the last descriptor in the first list.
6. When the last descriptor in the first linked list has been processed, an IRQ will be generated if the descriptor polling is disabled. Following this, your software needs to update the next descriptor pointer register with the address of the first descriptor in the second linked list before setting the run bit back to 1 to resume transfers. If descriptor polling is enabled, software does not need to update the next descriptor pointer register (for second descriptor linked list onwards) and set the run bit back to 1. These 2 steps are automatically done by hardware. The address of the new list is indicated by next descriptor pointer fields of the previous list. The Prefetcher core polls for the Owned by Hardware bit to be 1 in order to resume transfers. Software only needs to flip the Owned by Hardware bit of the first descriptor in second linked list to 1 to indicate to the Prefetcher core that the second linked list is ready.
7. If there are new descriptors to add, always add them to the list which the core is not processing (indicated by Owned By Hardware = 0). For example, if the core is processing the first list, add new descriptors to the second list and so forth. This method ensures that the descriptors are not updated when the core is processing them. Your software can read the descriptor in the memory to know the status of the transfer (for example; to know the actual bytes being transferred, any error in the transfer).

30.7.1.6.2. Resetting Prefetcher Core Flow

The following is the recommended flow for software to stop the mSGDMA when it is in the middle of operation.

1. Write 1 to the Prefetcher control register bit 2 (Reset_Prefetcher bit set to 1).
2. Poll for control register bit 2 to be 0 (Reset_Prefetcher bit cleared by hardware).
3. Trigger software reset condition in the dispatcher core.
4. Poll for software reset condition in the dispatcher core to be completed by reading the dispatcher core status register.
5. The whole reset flow has completed, software can reconfigure the mSGDMA.

30.7.1.7. Parameters

Table 308. Prefetcher Parameters

Name	Legal Value	Description
Enable Pre-fetching Module	1 or 0	1: Pre-fetching is enabled 0: Pre-fetching is disabled
Enable bursting on descriptor read host	1 or 0	1: Pre-fetching module uses Avalon-MM bursting when fetching descriptors.
Data Width (Avalon-MM Read/Write Descriptor)	32, 64, 128, 256, 512	Specifies the read and write data width of Avalon-MM read and write descriptor host.

continued...

Name	Legal Value	Description
Maximum Burst Count (Avalon-MM Read Descriptor)	1, 2, 4, 8, 16	Specifies the maximum read burst count of Avalon-MM read descriptor host.
Enable Extended Feature Support	1 or 0	This is a derived parameter from the mSGDMA top level composed. This is needed by this core to determine descriptor length (different length for standard/extended descriptor).
FIFO Depth	8, 16, 32, 64, 128, 256, 512, 1024	This is a derived parameter from the mSGDMA top level composed. This is needed by this core to determine its buffer depth to store next descriptor pointer information for descriptor write back.

30.8. Driver Implementation

Following section contains the APIs for the mSGDMA HAL Driver. An open mSGDMA API will instantiate an mSGDMA device with optional register interrupt service routine (ISR). You must define your own specific handling mechanism in the callback function when using an ISR. A callback function will be called by the ISR on error, early termination, and on transfer complete.

30.8.1. alt_msgdma_standard_descriptor_async_transfer

Table 309. alt_msgdma_standard_descriptor_async_transfer

Prototype:	int alt_msgdma_standard_descriptor_async_transfer(alt_msgdma_dev *dev, alt_msgdma_standard_descriptor *desc)
Include:	< modular_sgdma_dispatcher.h >, < altera_msgdma_csr_regs.h>, <altera_msgdma_descriptor_regs.h>, <sys/alt_errno.h>, <sys/alt_irq.h>, <io.h>
Parameters:	*dev — a pointer to msgdma instance. *desc — a pointer to a standard descriptor structure
Returns:	"0" for success, –ENOSPC indicates FIFO buffer is full, –EPERM indicates operation not permitted due to descriptor type conflict, -ETIME indicates Time out and skipping the looping after 5 msec.
Description:	A descriptor needs to be constructed and passing as a parameter pointer to *desc when calling this function. This function will call the helper function "alt_msgdma_descriptor_async_transfer" to start a non-blocking transfer of one standard descriptor at a time. If the FIFO buffer for a read/write is full at the time of this call, the routine will immediately return –ENOSPC, the application can then decide how to proceed without being blocked. -ETIME will be returned if the time spending for writing the descriptor to the dispatcher takes longer than 5 msec. You are advised to refer to the helper function for details. If a callback routine has been previously registered with this particular mSGDMA controller, the transfer will be set up to enable interrupt generation.

30.8.2. alt_msgdma_extended_descriptor_async_transfer

Table 310. alt_msgdma_extended_descriptor_async_transfer

Prototype:	int alt_msgdma_extended_descriptor_async_transfer(alt_msgdma_dev *dev, alt_msgdma_extended_descriptor *desc)
Include:	< modular_sgdma_dispatcher.h >, < altera_msgdma_csr_regs.h>, <altera_msgdma_descriptor_regs.h>, <sys/alt_errno.h>, <sys/alt_irq.h>, <io.h>
Parameters:	*dev — a pointer to mSGDMA instance. *desc — a pointer to an extended descriptor structure
Returns:	"0" for success, –ENOSPC indicates FIFO buffer is full, –EPERM indicates operation not permitted due to descriptor type conflict, -ETIME indicates time out and skipping the looping after 5 msec.
Description:	A descriptor needs to be constructed and passing as a parameter pointer to the *desc when calling this function. This function will call the helper function "alt_msgdma_descriptor_async_transfer" to start a non-blocking transfer of one standard descriptor at a time. If the FIFO buffer for a read/write is full at the time of this call, the routine will immediately return –ENOSPC, the application can then decide how to proceed without being blocked.-ETIME will be returned if the time spending for writing descriptor to the dispatcher takes longer than 5 msec. You are advised to refer the helper function for details. If a callback routine has been previously registered with this particular mSGDMA controller, the transfer will be set up to enable interrupt generation.

30.8.3. alt_msgdma_descriptor_async_transfer

Table 311. alt_msgdma_descriptor_async_transfer

Prototype:	static int alt_msgdma_descriptor_async_transfer(alt_msgdma_dev *dev, alt_msgdma_standard_descriptor *standard_desc, alt_msgdma_extended_descriptor *extended_desc)
Include:	< modular_sgdma_dispatcher.h >, < altera_msgdma_csr_regs.h>, <altera_msgdma_descriptor_regs.h>, <sys/alt_errno.h>, <sys/alt_irq.h>, <io.h>
Parameters:	*dev — a pointer to mSGDMA instance. *standard_desc — Pointer to single standard descriptor. *extended_desc — Pointer to single extended descriptor.
Returns:	"0" for success, –ENOSPC indicates FIFO buffer is full, –EPERM indicates operation not permitted due to descriptor type conflict, -ETIME indicates Time out and skipping the looping after 5 msec.
Description:	Helper functions for both "alt_msgdma_standard_descriptor_async_transfer" and "alt_msgdma_extended_descriptor_async_transfer". *Note:* Either one of both *standard_desc and *extended_desc must be assigned with NULL, another with proper pointer value. Failing to do so can cause the function return with "-EPERM ". If a callback routine has been previously registered with this particular mSGDMA controller, the transfer will be set up to enable interrupt generation. It is the responsibility of the application developer to check source interruption, status completion and creating suitable interrupt handling. *Note:* "stop on error" of the CSR control register is always masking within this function. The CSR control can be set by user through calling "alt_register_callback" with user defined control setting.

30.8.4. alt_msgdma_standard_descriptor_sync_transfer

Table 312. alt_msgdma_standard_descriptor_sync_transfer

Prototype:	int alt_msgdma_standard_descriptor_sync_transfer(alt_msgdma_dev *dev, alt_msgdma_standard_descriptor *desc)
Include:	< modular_sgdma_dispatcher.h >, < altera_msgdma_csr_regs.h>, <altera_msgdma_descriptor_regs.h>, <sys/alt_errno.h>, <sys/alt_irq.h>, <io.h>
Parameters:	*dev — a pointer to mSGDMA instance. *desc — a pointer to a standard descriptor structure
Returns:	"0" for success, "error" indicates errors or conditions causing msgdma stop issuing commands to hosts, suggest checking the bit set in the error with CSR status register."-EPERM" indicates operation not permitted due to descriptor type conflict. "-ETIME" indicates Time out and skipping the looping after 5 msec.
Description:	A standard descriptor needs to be constructed and passing as a parameter pointer to *desc when calling this function. This function will call helper function "alt_msgdma_descriptor_sync_transfer" to start a blocking transfer of one standard descriptor at a time. If the FIFO buffer for a read or write is full at the time of this call, the routine will wait until a free FIFO buffer is available to continue processing or a 5 msec time out. The function will return "error" if errors or conditions causing the dispatcher to stop issuing the commands to both the read and write hosts before both the read and write command buffers are empty. It is the responsibility of the application developer to check errors and completion status.

30.8.5. alt_msgdma_extended_descriptor_sync_transfer

Table 313. alt_msgdma_extended_descriptor_sync_transfer

Prototype:	int alt_msgdma_extended_descriptor_sync_transfer(alt_msgdma_dev *dev, alt_msgdma_extended_descriptor *desc)
Include:	< modular_sgdma_dispatcher.h >, < altera_msgdma_csr_regs.h>, <altera_msgdma_descriptor_regs.h>, <sys/alt_errno.h>, <sys/alt_irq.h>, <io.h>
Parameters:	*dev — a pointer to msgdma instance. *desc — a pointer to an extended descriptor structure
Returns:	"0" for success, "error" indicates errors or conditions causing msgdma stop issuing commands to hosts, suggest checking the bit set in the error with CSR status register."-EPERM" indicates operation not permitted due to descriptor type conflict. "-ETIME" indicates Time out and skipping the looping after 5 msec.
Description:	An extended descriptor needs to be constructed and passing as a parameter pointer to *desc when calling this function. This function will call helper function "alt_msgdma_descriptor_sync_transfer" to start-commencing a blocking transfer of one extended descriptor at a time. If the FIFO buffer for one of read or write is full at the time of this call, the routine will wait until free FIFO buffer available for continue processing or 5 msec time out. The function will return "error" if errors or conditions causing the dispatcher stop issuing the commands to both read and write hosts before both read and write command buffers are empty. It is the responsibility of the application developer to check errors and completion status. -ETIME will be returned if the time spending for waiting the FIFO buffer, writing descriptor to the dispatcher and any pending transfer to complete take longer than 5msec.

30.8.6. alt_msgdma_descriptor_sync_transfer

Table 314. alt_msgdma_descriptor_sync_transfer

Prototype:	int alt_msgdma_descriptor_sync_transfer(alt_msgdma_dev *dev, alt_msgdma_standard_descriptor *standard_desc, alt_msgdma_extended_descriptor *extended_desc)
Include:	< modular_sgdma_dispatcher.h >, < altera_msgdma_csr_regs.h>, <altera_msgdma_descriptor_regs.h>, <sys/alt_errno.h>, <sys/alt_irq.h>, <io.h>
Parameters:	*dev — a pointer to msgdma instance. *standard_desc — Pointer to single standard descriptor. *extended_desc — Pointer to single extended descriptor.
Returns:	"0" for success, "error" indicates errors or conditions causing msgdma stop issuing commands to hosts, suggest checking the bit set in the error with CSR status register."-EPERM" indicates operation not permitted due to descriptor type conflict. "-ETIME" indicates Time out and skipping the looping after 5 msec.
Description:	Helper functions for both "alt_msgdma_standard_descriptor_sync_transfer" and "alt_msgdma_extended_descriptor_sync_transfer". *Note:* Either one of both *standard_desc and *extended_desc must be assigned with NULL, another with proper pointer value. Failing to do so can cause the function return with "-EPERM . *Note:* "stop on error" of CSR control register is always being masked and the interrupt is always disabled within this function. The CSR control can be set by user through calling "alt_register_callback" with user defined control setting.

30.8.7. alt_msgdma_construct_standard_st_to_mm_descriptor

Table 315. alt_msgdma_construct_standard_st_to_mm_descriptor

Prototype:	int alt_msgdma_construct_standard_st_to_mm_descriptor (alt_msgdma_dev *dev, alt_msgdma_standard_descriptor *descriptor, alt_u32 *write_address, alt_u32 length, alt_u32 control)
Include:	< modular_sgdma_dispatcher.h >
Parameters:	*dev-a pointer to msgdma instance. *descriptor – a pointer to a standard descriptor structure. *write_address – a pointer to the base address of the destination memory. length - is used to specify the number of bytes to transfer per descriptor. The largest possible value can be filled in is "0Xffffffff". control – control field.
Returns:	"0" for success, -EINVAL for invalid argument, could be due to argument which has larger value than hardware setting value.
Description:	Function will call helper function "alt_msgdma_construct_standard_descriptor" for constructing st_to_mm standard descriptors. Unnecessary elements are set to 0 for completeness and will be ignored by the hardware.

30.8.8. alt_msgdma_construct_standard_mm_to_st_descriptor

Table 316. alt_msgdma_construct_standard_mm_to_st_descriptor

Prototype:	int alt_msgdma_construct_standard_mm_to_st_descriptor (alt_msgdma_dev *dev, alt_msgdma_standard_descriptor *descriptor, alt_u32 *read_address, alt_u32 length, alt_u32 control)
Include:	< modular_sgdma_dispatcher.h >
Parameters:	*dev-a pointer to msgdma instance. *descriptor – a pointer to a standard descriptor structure. *read_address – a pointer to the base address of the source memory. length – is used to specify the number of bytes to transfer per descriptor. The largest possible value can be filled in is "0Xffffffff". control – control field.
Returns:	"0" for success, -EINVAL for invalid argument, could be due to argument which has larger value than hardware setting value.
Description:	Function will call helper function "alt_msgdma_construct_standard_descriptor" for constructing mm_to_st standard descriptors. Unnecessary elements are set to 0 for completeness and will be ignored by the hardware.

30.8.9. alt_msgdma_construct_standard_mm_to_mm_descriptor

Table 317. alt_msgdma_construct_standard_mm_to_mm_descriptor

Prototype:	int alt_msgdma_construct_standard_mm_to_mm_descriptor (alt_msgdma_dev *dev, alt_msgdma_standard_descriptor *descriptor, alt_u32 *read_address, alt_u32 *write_address, alt_u32 length, alt_u32 control)
Include:	< modular_sgdma_dispatcher.h >
Parameters:	*dev-a pointer to msgdma instance. *descriptor – a pointer to a standard descriptor structure. *read_address – a pointer to the base address of the source memory. *write_address – a pointer to the base address of the destination memory. length – is used to specify the number of bytes to transfer per descriptor. The largest possible value can be filled in is "0Xffffffff". control – control field.
Returns:	"0" for success, -EINVAL for invalid argument, could be due to argument which has larger value than hardware setting value.
Description:	Function will call helper function "alt_msgdma_construct_standard_descriptor" for constructing mm_to_mm standard descriptors. Unnecessary elements are set to 0 for completeness and will be ignored by the hardware.

30.8.10. alt_msgdma_construct_standard_descriptor

Table 318. alt_msgdma_construct_standard_descriptor

Prototype:	static int alt_msgdma_construct_standard_descriptor (alt_msgdma_dev *dev, alt_msgdma_standard_descriptor *descriptor, alt_u32 *read_address, alt_u32 *write_address, alt_u32 length, alt_u32 control)
Include:	< modular_sgdma_dispatcher.h >
Parameters:	*dev-a pointer to msgdma instance. *descriptor – a pointer to a standard descriptor structure. *read_address – a pointer to the base address of the source memory. *write_address – a pointer to the base address of the destination memory. length - is used to specify the number of bytes to transfer per descriptor. The largest possible value can be filled in is "0Xffffffff". control – control field.
Returns:	"0" for success, -EINVAL for invalid argument, could be due to argument which has larger value than hardware setting value.
Description:	Helper functions for constructing mm_to_st, st_to_mm, mm_to_mm standard descriptors. Unnecessary elements are set to 0 for completeness and will be ignored by the hardware.

30.8.11. alt_msgdma_construct_extended_st_to_mm_descriptor

Table 319. alt_msgdma_construct_extended_st_to_mm_descriptor

Prototype:	int alt_msgdma_construct_extended_st_to_mm_descriptor (alt_msgdma_dev *dev, alt_msgdma_extended_descriptor *descriptor, alt_u32 *write_address, alt_u32 length, alt_u32 control, alt_u16 sequence_number, alt_u8 write_burst_count, alt_u16 write_stride)
Include:	< modular_sgdma_dispatcher.h >
Parameters:	*dev-a pointer to msgdma instance. *descriptor – a pointer to an extended descriptor structure. *write_address – a pointer to the base address of the destination memory. length – is used to specify the number of bytes to transfer per descriptor. The largest possible value can be filled in is "0Xffffffff". control – control field. sequence number – programmable sequence number to identify which descriptor has been sent to the host block. write_burst_count – programmable burst count between 1 and 128 and a power of 2. Setting to 0 will cause the host to use the maximum burst count instead. write_stride – programmable transfer stride. The stride value determines by how many words the host will increment the address. For fixed addresses the stride value is 0, sequential it is 1, every other word it is 2, etc…power of 2. Setting to 0 will cause the host to use the maximum burst count instead. write_stride – programmable transfer stride. The stride value determines by how many words the host will increment the address. For fixed addresses the stride value is 0, sequential it is 1, every other word it is 2, etc…
Returns:	"0" for success, -EINVAL for invalid argument, could be due to argument which has larger value than hardware setting value.
Description:	Function will call helper function "alt_msgdma_construct_extended_descriptor" for constructing st_to_mm extended descriptors. Unnecessary elements are set to 0 for completeness and will be ignored by the hardware.

30.8.12. alt_msgdma_construct_extended_mm_to_st_descriptor

Table 320. alt_msgdma_construct_extended_mm_to_st_descriptor

Prototype:	int alt_msgdma_construct_extended_mm_to_st_descriptor (alt_msgdma_dev *dev, alt_msgdma_extended_descriptor *descriptor, alt_u32 *read_address, alt_u32 length, alt_u32 control, alt_u16 sequence_number, alt_u8 read_burst_count, alt_u16 read_stride)
Include:	< modular_sgdma_dispatcher.h >
Parameters:	*dev-a pointer to msgdma instance. *descriptor – a pointer to an extended descriptor structure. *read_address – a pointer to the base address of the source memory. length – is used to specify the number of bytes to transfer per descriptor. The largest possible value can be filled in is "0Xffffffff". control – control field. sequence_number – programmable sequence number to identify which descriptor has been sent to the host block. read_burst_count – programmable burst count between 1 and 128 and a power of 2. Setting to 0 will cause the host to use the maximum burst count instead. read_stride – programmable transfer stride. The stride value determines by how many words the host will increment the address. For fixed addresses the stride value is 0, sequential it is 1, every other word it is 2, etc…
Returns:	"0" for success, -EINVAL for invalid argument, could be due to argument which has larger value than hardware setting value.
Description:	Function call helper function "alt_msgdma_construct_extended_descriptor" for constructing mm_to_st extended descriptors. Unnecessary elements are set to 0 for completeness and will be ignored by the hardware.

30.8.13. alt_msgdma_construct_extended_mm_to_mm_descriptor

Table 321. alt_msgdma_construct_extended_mm_to_mm_descriptor

Prototype:	int alt_msgdma_construct_extended_mm_to_mm_descriptor (alt_msgdma_dev *dev, alt_msgdma_extended_descriptor *descriptor, alt_u32 *read_address, alt_u32 *write_address, alt_u32 length, alt_u32 control, alt_u16 sequence_number, alt_u8 read_burst_count, alt_u8 write_burst_count, alt_u16 read_stride, alt_u16 write_stride)
Include:	< modular_sgdma_dispatcher.h >
Parameters:	*dev-a pointer to msgdma instance. *descriptor – a pointer to an extended descriptor structure. *read_address – a pointer to the base address of the source memory. *write_address – a pointer to the base address of the destination memory. length – is used to specify the number of bytes to transfer per descriptor. The largest possible value can be filled in is "0Xffffffff". control – control field. sequence_number – programmable sequence number to identify which descriptor has been sent to the host block. read_burst_count – programmable burst count between 1 and 128 and a power of 2. Setting to 0 will cause the host to use the maximum burst count instead. write_burst_count – programmable burst count between 1 and 128 and a power of 2. Setting to 0 will cause the host to use the maximum burst count instead. read_stride – programmable transfer stride. The stride value determines by how many words the host will increment the address. For fixed addresses the stride value is 0, sequential it is 1, ever other word it is 2, etc… write_stride – programmable transfer stride. The stride value determines by how many words the host will increment the address. For fixed addresses the stride value is 0, sequential it is 1, every other word it is 2, etc…
Returns:	"0" for success, -EINVAL for invalid argument, could be due to argument which has larger value than hardware setting value.
Description:	Function call helper function "alt_msgdma_construct_extended_descriptor" for constructing mm_to_mm extended descriptors. Unnecessary elements are set to 0 for completeness and will be ignored by the hardware.

30.8.14. alt_msgdma_construct_extended_descriptor

Table 322. alt_msgdma_construct_extended_descriptor

Prototype:	static int alt_msgdma_construct_descriptor (alt_msgdma_dev *dev, alt_msgdma_extended_descriptor *descriptor, alt_u32 *read_address, alt_u32 *write_address, alt_u32 length, alt_u32 control, alt_u16 sequence_number, alt_u8 read_burst_count, alt_u8 write_burst_count, alt_u16 read_stride, alt_u16 write_stride)
Include:	< modular_sgdma_dispatcher.h >
Parameters:	*dev-a pointer to msgdma instance. *descriptor – a pointer to an extended descriptor structure. *read_address – a pointer to the base address of the source memory. *write_address – a pointer to the base address of the destination memory. length – is used to specify the number of bytes to transfer per descriptor. The largest possible value can be filled in is "0Xffffffff". control – control field. sequence_number – programmable sequence number to identify which descriptor has been sent to the host block. read_burst_count – programmable burst count between 1 and 128 and a power of 2. Setting to 0 will cause the host to use the maximum burst count instead. write_burst_count – programmable burst count between 1 and 128 and a power of 2. Setting to 0 will cause the host to use the maximum burst count instead. read_stride – programmable transfer stride. The stride value determines by how many words the host will increment the address. For fixed addresses the stride value is 0, sequential it is 1, ever other word it is 2, etc… write_stride – programmable transfer stride. The stride value determines by how many words the host will increment the address. For fixed addresses the stride value is 0, sequential it is 1, every other word it is 2, etc…
Returns:	"0" for success, -EINVAL for invalid argument, could be due to argument which has larger value than hardware setting value.
Description:	Helper functions for constructing mm_to_st, st_to_mm, mm_to_mm extended descriptors. Unnecessary elements are set to 0 for completeness and will be ignored by the hardware.

30.8.15. alt_msgdma_register_callback

Table 323. alt_msgdma_register_callback

Prototype:	void alt_msgdma_register_callback(alt_msgdma_dev *dev, alt_msgdma_callback callback, alt_u32 control, void *context)
Include:	< modular_sgdma_dispatcher.h >
Parameters:	*dev — a pointer to msgdma instance. callback — Pointer to callback routine to execute at interrupt level control — Setting control register and OR with other control bits in the non_blocking and blocking transfer function. *context — pointer to user define context
Returns:	N/A
Description:	Associate a user-specific routine with the mSGDMA interrupt handler. If a callback is registered, all non-blocking mSGDMA transfers will enable interrupts that will cause the callback to be executed. The callback runs as part of the interrupt service routine, and great care must be taken to follow the guidelines for acceptable interrupt service routine behavior as described in the Nios II Software Developer's Handbook. However, user can change some of the CSR control setting in blocking transfer by calling this function.

30.8.16. alt_msgdma_open

Table 324. alt_msgdma_open

Prototype:	alt_msgdma_dev* alt_msgdma_open (const char* name)
Include:	< modular_sgdma_dispatcher.h >
Parameters:	*name — Character pointer to name of msgdma peripheral as registered with the HAL. For example, an mSGDMA in Platform Designer would be opened by asking for "MSGDMA_0_DISPATCHER_INTERNAL".
Returns:	Pointer to msgdma device instance struct, or null if the device. * could not be opened.
Description:	Retrieves a pointer to the mSGDMA instance.

30.8.17. alt_msgdma_write_standard_descriptor

Table 325. alt_msgdma_write_standard_descriptor

Prototype:	int alt_msgdma_write_standard_descriptor (alt_u32 csr_base, alt_u32 descriptor_base, alt_msgdma_standard_descriptor *descriptor)
Include:	< modular_sgdma_dispatcher.h >, <altera_msgdma_descriptor_regs.h>
Parameters:	csr_base – base address of the dispatcher CSR agent port. descriptor_base – base address of the dispatcher descriptor agent port. *descriptor – a pointer to a standard descriptor structure.
Returns:	Returns 0 upon success. Other return codes are defined in "alt_errno.h".
Description:	Sends a fully formed standard descriptor to the dispatcher module. If the dispatcher descriptor buffer is full, "-ENOSPC" is returned. This function is not reentrant since it must complete writing the entire descriptor to the dispatcher module and cannot be pre-empted.

30.8.18. alt_msgdma_write_extended_descriptor

Table 326. alt_msgdma_write_extended_descriptor

Prototype:	int alt_msgdma_write_extended_descriptor (alt_u32 csr_base, alt_u32 descriptor_base, alt_msgdma_extended_descriptor *descriptor)
Include:	< modular_sgdma_dispatcher.h >, <altera_msgdma_descriptor_regs.h>
Parameters:	csr_base – base address of the dispatcher CSR agent port. descriptor_base – base address of the dispatcher descriptor agent port. *descriptor – a pointer to an extended descriptor structure.
Returns:	Returns 0 upon success. Other return codes are defined in "alt_errno.h".
Description:	Sends a fully formed extended descriptor to the dispatcher module. If the dispatcher descriptor buffer is full an error is returned. This function is not reentrant since it must complete writing the entire descriptor to the dispatcher module and cannot be pre-empted.

30.8.19. alt_msgdma_init

Table 327. alt_msgdma_init

Prototype:	void alt_msgdma_init (alt_msgdma_dev *dev, alt_u32 ic_id, alt_u32 irq)
Include:	< modular_sgdma_dispatcher.h >, <altera_msgdma_descriptor_regs.h>, <altera_msgdma_csr_regs.h>
Parameters:	*dev – a pointer to mSGDMA instance. ic_id – id of irq interrupt controller irq – irq number that belonged to mSGDMA instance
Returns:	N/A
Description:	Initializes the mSGDMA controller. This routine is called from the ALTERA_AVALON_MSGDMA_INIT macro and is called automatically by "alt_sys_init.c".

30.8.20. alt_msgdma_irq

Table 328. alt_msgdma_irq

Prototype:	void alt_msgdma_irq(void *context)
Include:	< modular_sgdma_dispatcher.h >, <sys/alt_irq.h>, <altera_msgdma_csr_regs.h>
Parameters:	*context – a pointer to mSGDMA instance.
Returns:	N/A
Description:	Interrupt handler for mSGDMA. This function will call the user defined interrupt handler if user registers their own interrupt handler with calling “alt_register_callback”.

30.9. Example Code Using mSGDMA Core

Refer to the mSGDMA design example that showcases the use of mSGDMA core to transfer memory content from one location to another.

30.10. Modular Scatter-Gather DMA Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2020.09.21	20.2	Corrected link to the *mSGDMA design example*.
2019.04.01	19.1	• Updated the design example information in section: *Example Code Using mSGDMA Core*. • Added support for write response: — Added parameter: **Enable Write Response**. — Added descriptor bit: `Wait for write responses`.
2018.09.24	18.1	Added a new section *Example Code Using mSGDMA Core*.
2018.05.07	18.0	• Added a new parameter **No Byteenables During Writes** in the *Table: Component Parameters*. • Corrected information in section *Register Map of mSGDMA*. • Added register details for the following: — Write Fill Level — Read Fill Level — Response Fill Level — Write Sequence Number — Read Sequence Number — Component Configuration 1 — Component Configuration 2 — Component Version Register — Component Type Register

Date	Version	Changes
May 2017	2017.05.08	Status Register on page 343 : Bit 9 description updated
May 2016	2016.05.03	Updated tables: • Component Parameters
December 2015	2015.12.16	Added "alt_msgdma_irq" section.
November 2015	2015.11.06	Updated sections: • Response Port • Component Parameters Sections added: • Programming Model — Stop DMA Operation — Stop Descriptor Operation — Recovery from Stopped on Error and Stopped on Early Termination • Modular Scatter-Gather DMA Prefetcher Core • Driver Implementation Section removed: • Unsupported Feature
July 2014	2014.07.24	Initial release

31. Scatter-Gather DMA Controller Core

Intel recommends to use Modular Scatter-Gather DMA Core for your new designs.

31.1. Core Overview

The Scatter-Gather Direct Memory Access (SG-DMA) controller core implements high-speed data transfer between two components. You can use the SG-DMA controller core to transfer data from:

- Memory to memory
- Data stream to memory
- Memory to data stream

The SG-DMA controller core transfers and merges non-contiguous memory to a continuous address space, and vice versa. The core reads a series of descriptors that specify the data to be transferred.

For applications requiring more than one DMA channel, multiple instantiations of the core can provide the required throughput. Each SG-DMA controller has its own series of descriptors specifying the data transfers. A single software module controls all of the DMA channels.

For the Nios II processor, device drivers are provided in the Hardware Abstraction Layer (HAL) system library, allowing software to access the core using the provided driver.

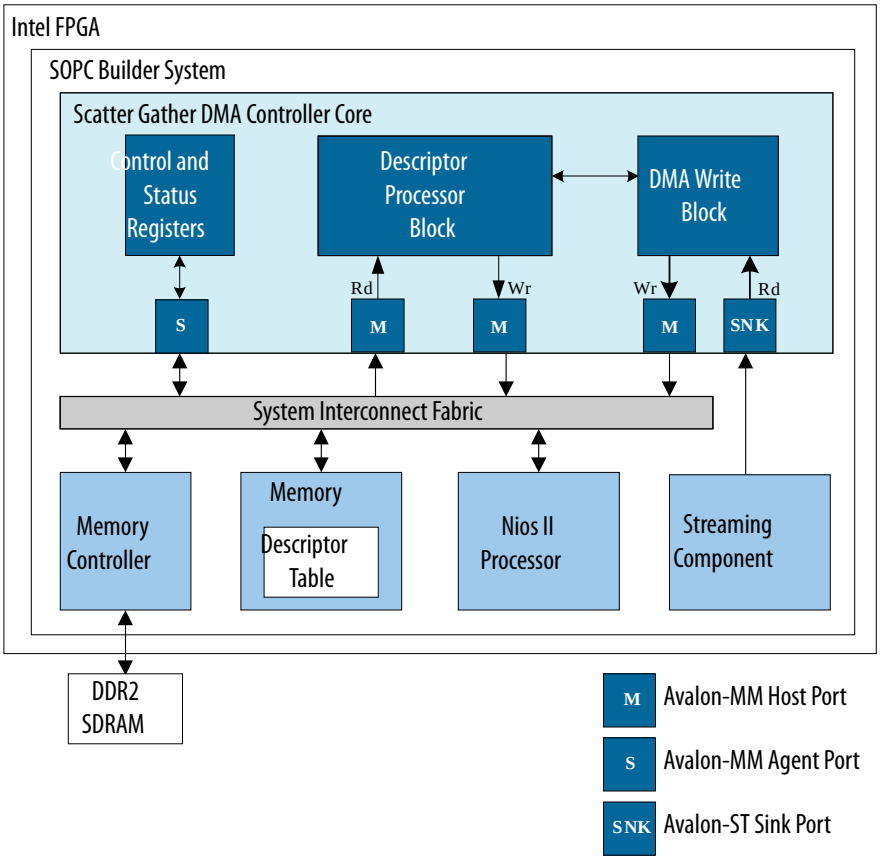
31.1.1. Example Systems

The block diagram below shows a SG-DMA controller core for the DMA subsystem of a printed circuit board. The SG-DMA core in the FPGA reads streaming data from an internal streaming component and writes data to an external memory. A Nios II processor provides overall system control.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

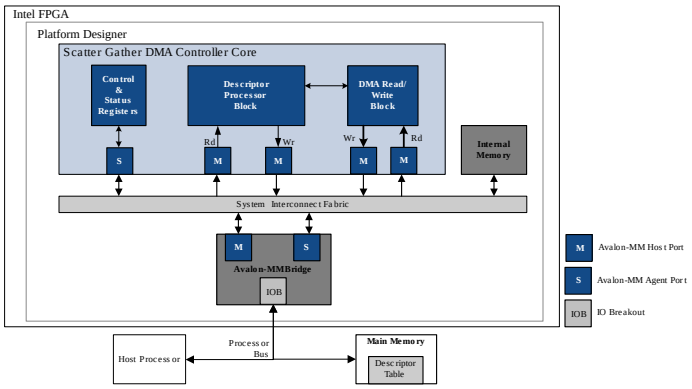
ISO 9001:2015 Registered

Figure 91. SG-DMA Controller Core with Streaming Peripheral and External Memory



The figure below shows a different use of the SG-DMA controller core, where the core transfers data between an internal and external memory. The host processor and memory are connected to a system bus, typically either a PCI Express or Serial RapidIO™.

Figure 92. SG-DMA Controller Core with Internal and External Memory



31.1.2. Comparison of SG-DMA Controller Core and DMA Controller Core

The SG-DMA controller core provides a significant performance enhancement over the previously available DMA controller core, which could only queue one transfer at a time. Using the DMA Controller core, a CPU had to wait for the transfer to complete before writing a new descriptor to the DMA agent port. Transfers to non-contiguous memory could not be linked; consequently, the CPU overhead was substantial for small transfers, degrading overall system performance. In contrast, the SG-DMA controller core reads a series of descriptors from memory that describe the required transactions and performs all of the transfers without additional intervention from the CPU.

31.2. Resource Usage and Performance

Resource utilization for the core is 600–1400 logic elements, depending upon the width of the datapath, the parameterization of the core, the device family, and the type of data transfer. The table below provides the estimated resource usage for a SG-DMA controller core used for memory to memory transfer. The core is configurable and the resource utilization varies with the configuration specified.

Table 329. SG-DMA Estimated Resource Usage

Datapath	Cyclone II	Stratix> (LEs)	Stratix II (ALUTs)
8-bit datapath	850	650	600
32-bit datapath	1100	850	700
64-bit datapath	1250	1250	800

The core operating frequency varies with the device and the size of the datapath. The table below provides an example of expected performance for SG-DMA cores instantiated in several different device families.

Table 330. SG-DMA Peak Performance

Device	Datapath	f_{MAX}	Throughput
Cyclone II	64 bits	150 MHz	9.6 Gbps
Cyclone III	64 bits	160 MHz	10.2 Gbps
Stratix II/Stratix II GX	64 bits	250 MHz	16.0 Gbps
Stratix III	64 bits	300 MHz	19.2 Gbps

31.3. Functional Description

The SG-DMA controller core comprises three major blocks: descriptor processor, DMA read, and DMA write. These blocks are combined to create three different configurations:

- Memory to memory
- Memory to stream
- Stream to memory

  The type of devices you are transferring data to and from determines the configuration to implement. Examples of memory-mapped devices are PCI, PCIe and most memory devices. The Triple Speed Ethernet MAC, DSP IP core and many video IPs are examples of streaming devices. A recompilation is necessary each time you change the configuration of the SG-DMA controller core.

31.3.1. Functional Blocks and Configurations

The following sections describe each functional block and configuration.

Descriptor Processor

The descriptor processor reads descriptors from the descriptor list via its Avalon Memory-Mapped (MM) read host port and pushes commands into the command FIFOs of the DMA read and write blocks. Each command includes the following fields to specify a transfer:

- Source address
- Destination address
- Number of bytes to transfer
- Increment read address after each transfer
- Increment write address after each transfer
- Generate start of packet (SOP) and end of packet (EOP)

  After each command is processed by the DMA read or write block, a status token containing information about the transfer such as the number of bytes actually written is returned to the descriptor processor, where it is written to the respective fields in the descriptor.

DMA Read Block

The DMA read block is used in memory-to-memory and memory-to-stream configurations. The block performs the following operations:

- Reads commands from the input command FIFO.
- Reads a block of memory via the Avalon-MM read host port for each command.
- Pushes data into the data FIFO.

  If burst transfer is enabled, an internal read FIFO with a depth of twice the maximum read burst size is instantiated. The DMA read block initiates burst reads only when the read FIFO has sufficient space to buffer the complete burst.

DMA Write Block

The DMA write block is used in memory-to-memory and stream-to-memory configurations. The block reads commands from its input command FIFO. For each command, the DMA write block reads data from its Avalon-ST sink port and writes it to the Avalon-MM host port.

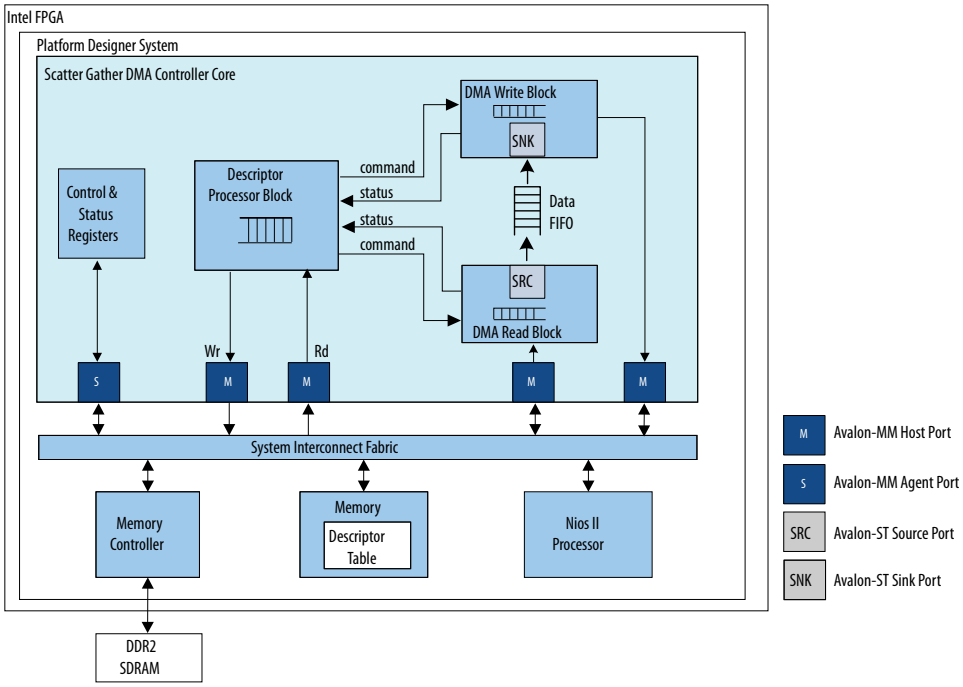
If burst transfer is enabled, an internal write FIFO with a depth of twice the maximum write burst size is instantiated. Each burst write transfers a fixed amount of data equals to the write burst size, except for the last burst. In the last burst, the remaining data is transferred even if the amount of data is less than the write burst size.

Memory-to-Memory Configuration

Memory-to-memory configurations include all three blocks: descriptor processor, DMA read, and DMA write. An internal FIFO is also included to provide buffering and flow control for data transferred between the DMA read and write blocks.

The example below illustrates one possible memory-to-memory configuration with an internal Nios II processor and descriptor list.

Figure 93. Example of Memory-to-Memory Configuration

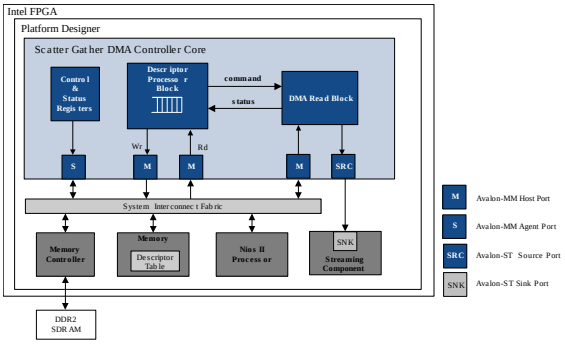


Memory-to-Stream Configuration

Memory-to-stream configurations include the descriptor processor and DMA read blocks.

In this example, the Nios II processor and descriptor table are in the FPGA. Data from an external DDR2 SDRAM is read by the SG-DMA controller and written to an on-chip streaming peripheral.

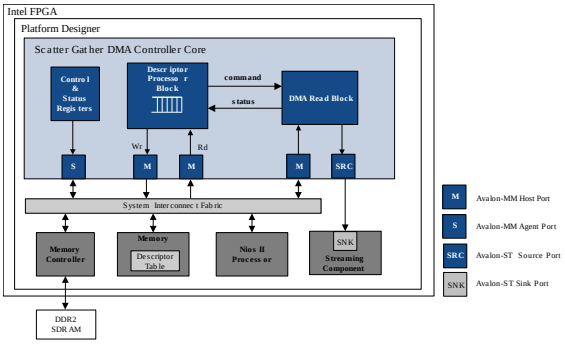
Figure 94. Example of Memory-to-Stream Configuration



Stream-to-Memory Configuration

Stream-to-memory configurations include the descriptor processor and DMA write blocks. This configuration is similar to the memory-to-stream configuration as the figure below illustrates.

Figure 95. Example of Stream-to-Memory Configuration



31.3.2. DMA Descriptors

DMA descriptors specify data transfers to be performed. The SG-DMA core uses a dedicated interface to read and write the descriptors. These descriptors, which are stored as a linked list, can be stored on an on-chip or off-chip memory and can be arbitrarily long.

Storing the descriptor list in an external memory frees up resources in the FPGA; however, an external descriptor list increases the overhead involved when the descriptor processor reads and updates the list. The SG-DMA core has an internal FIFO to store descriptors read from memory, which allows the core to perform descriptor read, execute, and write back operations in parallel, hiding the descriptor access and processing overhead.

The descriptors must be initialized and aligned on a 32-bit boundary. The last descriptor in the list must have its `OWNED_BY_HW` bit set to 0 because the core relies on a cleared `OWNED_BY_HW` bit to stop processing.

See the **DMA Descriptors** section for the structure of the DMA descriptor.

Descriptor Processing

The following steps describe how the DMA descriptors are processed:

1. Software builds the descriptor linked list. See the **Building and Updating Descriptors List** section for more information on how to build and update the descriptor linked list.
2. Software writes the address of the first descriptor to the `next_descriptor_pointer` register and initiates the transfer by setting the RUN bit in the `control` register to 1. See the **Software Programming Model** section for more information on the registers.

   On the next clock cycle following the assertion of the RUN bit, the core sets the BUSY bit in the `status` register to 1 to indicate that descriptor processing is executing.
3. The descriptor processor block reads the address of the first descriptor from the `next_descriptor_pointer` register and pushes the retrieved descriptor into the command FIFO, which feeds commands to both the DMA read and write blocks. As soon as the first descriptor is read, the block reads the next descriptor and pushes it into the command FIFO. One descriptor is always read in advance thus maximizing throughput.
4. The core performs the data transfer.
   - In memory-to-memory configurations, the DMA read block receives the source address from its command FIFO and starts reading data to fill the FIFO on its stream port until the specified number of bytes are transferred. The DMA read block pauses when the FIFO is full until the FIFO has enough space to accept more data.

     The DMA write block gets the destination address from its command FIFO and starts writing until the specified number of bytes are transferred. If the data FIFO ever empties, the write block pauses until the FIFO has more data to write.
   - In memory-to-stream configurations, the DMA read block reads from the source address and transfers the data to the core's streaming port until the specified number of bytes are transferred or the end of packet is reached. The block uses the end-of-packet indicator for transfers with an unknown transfer size. For data transfers without using the end-of-packet indicator, the transfer size must be a multiple of the data width. Otherwise, the block requires extra logic and may impact the system performance.
   - In stream-to-memory configurations, the DMA write block reads from the core's streaming port and writes to the destination address. The block continues reading until the specified number of bytes are transferred.
5. The descriptor processor block receives a status from the DMA read or write block and updates the DESC_CONTROL, DESC_STATUS, and ACTUAL_BYTES_TRANSFERRED fields in the descriptor. The OWNED_BY_HW bit in the DESC_CONTROL field is cleared unless the PARK bit is set to 1.

   Once the core starts processing the descriptors, software must not update descriptors with OWNED_BY_HW bit set to 1. It is only safe for software to update a descriptor when its OWNED_BY_HW bit is cleared.

The SG-DMA core continues processing the descriptors until an error condition occurs and the STOP_DMA_ER bit is set to 1, or a descriptor with a cleared OWNED_BY_HW bit is encountered.

Building and Updating Descriptor List

Intel recommends the following method of building and updating the descriptor list:

1. Build the descriptor list and terminate the list with a non-hardware owned descriptor (OWNED_BY_HW = 0). The list can be arbitrarily long.
2. Set the interrupt IE_CHAIN_COMPLETED.
3. Write the address of the first descriptor in the first list to the next_descriptor_pointer register and set the RUN bit to 1 to initiate transfers.
4. While the core is processing the first list, build a second list of descriptors.
5. When the SD-DMA controller core finishes processing the first list, an interrupt is generated. Update the next_descriptor_pointer register with the address of the first descriptor in the second list. Clear the RUN bit and the status register. Set the RUN bit back to 1 to resume transfers.
6. If there are new descriptors to add, always add them to the list which the core is not processing. For example, if the core is processing the first list, add new descriptors to the second list and so forth.

   This method ensures that the descriptors are not updated when the core is processing them. Because the method requires a response to the interrupt, a high-latency interrupt may cause a problem in systems where stalling data movement is not possible.

31.3.3. Error Conditions

The SG-DMA core has a configurable error width. Error signals are connected directly to the Avalon-ST source or sink to which the SG-DMA core is connected.

The list below describes how the error signals in the SG-DMA core are implemented in the following configurations:

- Memory-to-memory configuration

  No error signals are generated. The error field in the register and descriptor is hardcoded to 0.

- Memory-to-stream configuration

  If you specified the usage of error bits in the core, the error bits are generated in the Avalon-ST source interface. These error bits are hardcoded to 0 and generated in compliance with the Avalon-ST agent interfaces.

- Stream-to-memory configuration

  If you specified the usage of error bits in the core, error bits are generated in the Avalon-ST sink interface. These error bits are passed from the Avalon-ST sink interface and stored in the registers and descriptor.

  The table below lists the error signals when the core is operating in the memory-to-stream configuration and connected to the transmit FIFO interface of the Intel FPGA Triple-Speed Ethernet IP core.

Table 331. Avalon-ST Transmit Error Types

Signal Type	Description
TSE_transmit_error[0]	Transmit Frame Error. Asserted to indicate that the transmitted frame should be viewed as invalid by the Ethernet MAC. The frame is then transferred onto the GMII interface with an error code during the frame transfer.

The table below lists the error signals when the core is operating in the stream-to-memory configuration and connected to the transmit FIFO interface of the Triple-Speed Ethernet IP Core.

Table 332. Avalon-ST Receive Error Types

Signal Type	Description
TSE_receive_error[0]	Receive Frame Error. This signal indicates that an error has occurred. It is the logical OR of receive errors 1 through 5.
TSE_receive_error[1]	Invalid Length Error. Asserted when the received frame has an invalid length as defined by the IEEE 802.3 standard.
TSE_receive_error[2]	CRC Error. Asserted when the frame has been received with a CRC-32 error.
TSE_receive_error[3]	Receive Frame Truncated. Asserted when the received frame has been truncated due to receive FIFO overflow.
TSE_receive_error[4]	Received Frame corrupted due to PHY error. (The PHY has asserted an error on the receive GMII interface.)
TSE_receive_error[5]	Collision Error. Asserted when the frame was received with a collision.

Each streaming core has a different set of error codes. Refer to the respective user guides for the codes.

31.4. Parameters

Table 333. Configurable Parameters

Parameter	Legal Values	Description
Transfer mode	Memory To Memory Memory To Stream Stream To Memory	Configuration to use. For more information about these configurations, see the **Memory-to-Memory Configuration** section.
Enable bursting on descriptor read host	On/Off	If this option is on, the descriptor processor block uses Avalon-MM bursting when fetching descriptors and writing them back in memory. With 32-bit read and write ports, the descriptor processor block can fetch the 256-bit descriptor by performing 8-word burst as opposed to eight individual single-word transactions.
Allow unaligned transfers	On/Off	If this option is on, the core allows accesses to non-word-aligned addresses. This option doesn't apply for burst transfers. Unaligned transfers require extra logic that may negatively impact system performance.
Enable burst transfers	On/Off	Turning on this option enables burst reads and writes.
Read burstcount signal width	1–16	The width of the read `burstcount` signal. This value determines the maximum burst read size.
Write burstcount signal width	1–16	The width of the write `burstcount` signal. This value determines the maximum burst write size.
Data width	8, 16, 32, 64	The data width in bits for the Avalon-MM read and write ports.
Source error width	0–7	The width of the `error` signal for the Avalon-ST source port.
Sink error width	0 – 7	The width of the `error` signal for the Avalon-ST sink port.
Data transfer FIFO depth	2, 4, 8, 16, 32, 64	The depth of the internal data FIFO in memory-to-memory configurations with burst transfers disabled.

The SG-DMA controller core should be given a higher priority (lower IRQ value) than most of the components in a system to ensure high throughput.

31.5. Simulation Considerations

Signals for hardware simulation are automatically generated as part of the Nios II simulation process available in the Nios II IDE.

31.6. Software Programming Model

The following sections describe the software programming model for the SG-DMA controller core.

31.6.1. HAL System Library Support

The Intel-provided driver implements a HAL device driver that integrates into the HAL system library for Nios II systems. HAL users should access the SG-DMA controller core via the familiar HAL API and the ANSI C standard library.

31.6.2. Software Files

The SG-DMA controller core provides the following software files. These files provide low-level access to the hardware and drivers that integrate into the Nios II HAL system library. Application developers should not modify these files.

- `altera_avalon_sgdma_regs.h`—defines the core's register map, providing symbolic constants to access the low-level hardware
- `altera_avalon_sgdma.h`—provides definitions for the Intel FPGA Avalon SG-DMA buffer control and status flags.
- `altera_avalon_sgdma.c`—provides function definitions for the code that implements the SG-DMA controller core.
- `altera_avalon_sgdma_descriptor.h`—defines the core's descriptor, providing symbolic constants to access the low-level hardware.

31.6.3. Register Maps

The SG-DMA controller core has three registers accessible from its Avalon-MM interface; `status`, `control` and `next_descriptor_pointer`. Software can configure the core and determines its current status by accessing the registers.

The `control`/`status` register has a 32-bit interface without byte-enable logic, and therefore cannot be properly accessed by a host with narrower data width than itself. To ensure correct operation of the core, always access the register with a host that is at least 32 bits wide.

Table 334. Register Map

32-bit Word Offset (Byte Offset)	Register Name	Reset Value	Description
base + 0 (0x0)	`status`	0	This register indicates the core's current status such as what caused the last interrupt and if the core is still processing descriptors. See the **status Register Map** table for the `status` register map.
base + 1 (0x4)	`version`	1	Indicate the hardware version number. Only being used by software driver for software backward compatibility purpose.
base + 4 (0x10)	`control`	0	This register specifies the core's behavior such as what triggers an interrupt and when the core is started and stopped. The host processor can configure the core by setting the register bits accordingly. See the **Control Register Bit Map** table for the `control` register map.
base + 8 (0x20)	`next_descriptor_pointer`	0	This register contains the address of the next descriptor to process. Set this register to the address of the first descriptor as part of the system initialization sequence. Intel recommends that user applications clear the RUN bit in the `control` register and wait until the BUSY bit of the `status` register is set to 0 before reading this register.

Table 335. Control Register Bit Map

Bit	Bit Name	Access	Description
0	IE_ERROR	R/W	When this bit is set to 1, the core generates an interrupt if an Avalon-ST error occurs during descriptor processing. (1)
1	IE_EOP_ENCOUNTERED	R/W	When this bit is set to 1, the core generates an interrupt if an EOP is encountered during descriptor processing. (1)
2	IE_DESCRIPTOR_COMPLETED	R/W	When this bit is set to 1, the core generates an interrupt after each descriptor is processed. (1)
3	IE_CHAIN_COMPLETED	R/W	When this bit is set to 1, the core generates an interrupt after the last descriptor in the list is processed, that is when the core encounters a descriptor with a cleared OWNED_BY_HW bit. (1)
4	IE_GLOBAL	R/W	When this bit is set to 1, the core is enabled to generate interrupts.
5	RUN	R/W	Set this bit to 1 to start the descriptor processor block which subsequently initiates DMA transactions. Prior to setting this bit to 1, ensure that the register next_descriptor_pointer is updated with the address of the first descriptor to process. The core continues to process descriptors in its queue as long as this bit is 1. Clear this bit to stop the core from processing the next descriptor in its queue. If this bit is cleared in the middle of processing a descriptor, the core completes the processing before stopping. The host processor can then modify the remaining descriptors and restart the core.
6	STOP_DMA_ER	R/W	Set this bit to 1 to stop the core when an Avalon-ST error is encountered during a DMA transaction. This bit applies only to stream-to-memory configurations.
7	IE_MAX_DESC_PROCESSED	R/W	Set this bit to 1 to generate an interrupt after the number of descriptors specified by MAX_DESC_PROCESSED are processed.
8 .. 15	MAX_DESC_PROCESSED	R/W	Specifies the number of descriptors to process before the core generates an interrupt.
16	SW_RESET	R/W	Software can reset the core by writing to this bit twice. Upon the second write, the core is reset. The logic which sequences the software reset process then resets itself automatically. Executing a software reset when a DMA transfer is active may result in permanent bus lockup until the next system reset. Hence, Intel recommends that you use the software reset as your last resort.
17	PARK	R/W	Setting this bit to 0 causes the SG-DMA controller core to clear the OWNED_BY_HW bit in the descriptor after each descriptor is processed. If the PARK bit is set to 1, the core does not clear the OWNED_BY_HW bit, thus allowing the same descriptor to be processed repeatedly without software intervention. You also need to set the last descriptor in the list to point to the first one.
18	DESC_POLL_EN	R/W	Set this bit to 1 to enable polling mode. When you set this bit to 1, the core continues to poll for the next descriptor until the OWNED_BY_HW bit is set. The core also updates the descriptor pointer to point to the current descriptor.
19	Reserved		
20..30	TIMEOUT_COUNTER	RW	Specifies the number of clocks to wait before polling again. The valid range is 1 to 255. The core also updates the next_desc_ptr field so that it points to the next descriptor to read.
31	CLEAR_INTERRUPT	R/W	Set this bit to 1 to clear pending interrupts.

Note :
1. All interrupts are generated only after the descriptor is updated.

Intel recommends that you read the status register only after the RUN bit in the `control` register is cleared.

Table 336. Status Register Bit Map

Bit	Bit Name	Access	Description
0	ERROR	R/C (1) (2)	A value of 1 indicates that an Avalon-ST error was encountered during a transfer.
1	EOP_ENCOUNTERED	R/C	A value of 1 indicates that the transfer was terminated by an end-of-packet (EOP) signal generated on the Avalon-ST source interface. This condition is only possible in stream-to-memory configurations.
2	DESCRIPTOR_COMPLETED	R/C (1) (2)	A value of 1 indicates that a descriptor was processed to completion.
3	CHAIN_COMPLETED	R/C (1) (2)	A value of 1 indicates that the core has completed processing the descriptor chain.
4	BUSY	R (3)	A value of 1 indicates that descriptors are being processed. This bit is set to 1 on the next clock cycle after the RUN bit is asserted and does not get cleared until one of the following event occurs: • After the processing of the descriptor completes and the RUN bit is cleared. • When an error condition occurs, the STOP_DMA_ER bit is set to 1 and the processing of the current descriptor completes.
5 .. 31	Reserved		

Note :
1. This bit must be cleared after a read is performed. Write one to clear this bit.
2. This bit is updated by hardware after each DMA transfer completes. It remains set until software writes one to clear.
3. This bit is continuously updated by the hardware.

31.6.4. DMA Descriptors

See the **Data Structure** section for the structure definition.

Table 337. DMA Descriptor Structure

Byte Offset	Field Names											
	31		24	23		16	15		8	7		0
base	source											
base + 4	Reserved											
base + 8	destination											
base + 12	Reserved											
base + 16	next_desc_ptr											
base + 20	Reserved											
base + 24	Reserved						bytes_to_transfer					
base + 28	desc_control			desc_status			actual_bytes_transferred					

Table 338. DMA Descriptor Field Description

Field Name	Access	Description
source	R/W	Specifies the address of data to be read. This address is set to 0 if the input interface is an Avalon-ST interface.
destination	R/W	Specifies the address to which data should be written. This address is set to 0 if the output interface is an Avalon-ST interface.
next_desc_ptr	R/W	Specifies the address of the next descriptor in the linked list.
bytes_to_transfer	R/W	Specifies the number of bytes to transfer. If this field is 0, the SG-DMA controller core continues transferring data until it encounters an EOP.
actual_bytes_transferred	R	Specifies the number of bytes that are successfully transferred by the core. This field is updated after the core processes a descriptor.
desc_status	R/W	This field is updated after the core processes a descriptor. See **DESC_STATUS Bit Map** for the bit map of this field.
desc_control	R/W	Specifies the behavior of the core. This field is updated after the core processes a descriptor. See the **DESC_CONTROL Bit Map** table for descriptions of each bit.

Table 339. DESC_CONTROL Bit Map

Bit (s)	Field Name	Access	Description
0	GENERATE_EOP	W	When this bit is set to 1,the DMA read block asserts the EOP signal on the final word.
1	READ_FIXED_ADDRESS	R/W	This bit applies only to Avalon-MM read host ports. When this bit is set to 1, the DMA read block does not increment the memory address. When this bit is set to 0, the read address increments after each read.
2	WRITE_FIXED_ADDRESS	R/W	This bit applies only to Avalon-MM write host ports. When this bit is set to 1, the DMA write block does not increment the memory address. When this bit is set to 0, the write address increments after each write. In memory-to-stream configurations, the DMA read block generates a start-of-packet (SOP) on the first word when this bit is set to 1.
[6:3]	Reserved	—	—
3 .. 6	AVALON-ST_CHANNEL_NUMBER	R/W	The DMA read block sets the channel signal to this value for each word in the transaction. The DMA write block replaces this value with the channel number on its sink port.
7	OWNED_BY_HW	R/W	This bit determines whether hardware or software has write access to the current register. When this bit is set to 1, the core can update the descriptor and software should not access the descriptor due to the possibility of race conditions. Otherwise, it is safe for software to update the descriptor.

After completing a DMA transaction, the descriptor processor block updates the desc_status field to indicate how the transaction proceeded.

Table 340. DESC_STATUS Bit Map

Bit	Bit Name	Access	Description
[7:0]	ERROR_0 .. ERROR_7	R	Each bit represents an error that occurred on the Avalon-ST interface. The context of each error is defined by the component connected to the Avalon-ST interface.

31.6.5. Timeouts

The SG-DMA controller does not implement internal counters to detect stalls. Software can instantiate a timer component if this functionality is required.

31.7. Programming with SG-DMA Controller

This section describes the device and descriptor data structures, and the application programming interface (API) for the SG-DMA controller core.

31.7.1. Data Structure

Table 341. Device Data Structure

```
typedef struct alt_sgdma_dev
{
alt_llist llist; // Device linked-list entry
const char *name; // Name of SGDMA in SOPC System
void *base; // Base address of SGDMA
alt_u32 *descriptor_base; // reserved
alt_u32 next_index; // reserved
alt_u32 num_descriptors; // reserved
alt_sgdma_descriptor *current_descriptor; // reserved
alt_sgdma_descriptor *next_descriptor; // reserved
alt_avalon_sgdma_callback callback; // Callback routine pointer
void *callback_context; // Callback context pointer
alt_u32 chain_control; // Value OR'd into control reg
} alt_sgdma_dev;
```

Table 342. Descriptor Data Structure

```
typedef struct {
alt_u32 *read_addr;
alt_u32 read_addr_pad;
alt_u32 *write_addr;
alt_u32 write_addr_pad;
alt_u32 *next;
alt_u32 next_pad;
alt_u16 bytes_to_transfer;
alt_u8 read_burst; /* Reserved field. Set to 0. */
alt_u8 write_burst;/* Reserved field. Set to 0. */
alt_u16 actual_bytes_transferred;
alt_u8 status;
alt_u8 control;
} alt_avalon_sgdma_packed alt_sgdma_descriptor;
```

31.7.2. SG-DMA API

Table 343. Function List

Name	Description
`alt_avalon_sgdma_do_async_transfer()`	Starts a non-blocking transfer of a descriptor chain.
`alt_avalon_sgdma_do_sync_transfer()`	Starts a blocking transfer of a descriptor chain. This function blocks both before transfer if the controller is busy and until the requested transfer has completed.
`alt_avalon_sgdma_construct_mem_to__mem_desc()`	Constructs a single SG-DMA descriptor in the specified memory for an Avalon-MM to Avalon-MM transfer.
`alt_avalon_sgdma_construct_stream_to_mem_desc()`	Constructs a single SG-DMA descriptor in the specified memory for an Avalon-ST to Avalon-MM transfer. The function automatically terminates the descriptor chain with a NULL descriptor.
`alt_avalon_sgdma_construct_mem_to_stream_desc()`	Constructs a single SG-DMA descriptor in the specified memory for an Avalon-MM to Avalon-ST transfer.
`alt_avalon_sgdma_enable_desc_poll()`	Enables descriptor polling mode. To use this feature, you need to make sure that the hardware supports polling.
`alt_avalon_sgdma_disable_desc_poll()`	Disables descriptor polling mode.
`alt_avalon_sgdma_check_descriptor_status()`	Reads the status of a given descriptor.
`alt_avalon_sgdma_register_callback()`	Associates a user-specific callback routine with the SG-DMA interrupt handler.
`alt_avalon_sgdma_start()`	Starts the DMA engine. This is not required when `alt_avalon_sgdma_do_async_transfer()`and `alt_avalon_sgdma_do_sync_transfer()` are used.
`alt_avalon_sgdma_stop()`	Stops the DMA engine. This is not required when `alt_avalon_sgdma_do_async_transfer()`and `alt_avalon_sgdma_do_sync_transfer()` are used.
`alt_avalon_sgdma_open()`	Returns a pointer to the SG-DMA controller with the given name.

31.7.3. alt_avalon_sgdma_do_async_transfer()

Prototype:	int alt_avalon_do_async_transfer(alt_sgdma_dev *dev, alt_sgdma_descriptor *desc)
Thread-safe:	No.
Available from ISR:	Yes.
Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs**.h>
Parameters:	*dev—a pointer to an SG-DMA device structure. *desc—a pointer to a single, constructed descriptor. The descriptor must have its "next" descriptor field initialized either to a non-ready descriptor, or to the next descriptor in the chain.
Returns:	Returns 0 success. Other return codes are defined in **errno.h**.
Description:	Set up and begin a non-blocking transfer of one or more descriptors or a descriptor chain. If the SG-DMA controller is busy at the time of this call, the routine immediately returns EBUSY; the application can then decide how to proceed without being blocked. If a callback routine has been previously registered with this particular SG-DMA controller, the transfer is set up to issue an interrupt on error, EOP, or chain completion. Otherwise, no interrupt is registered and the application developer must check for and handle errors and completion. The run bit is cleared before the beginning of the transfer and is set to 1 to restart a new descriptor chain.

31.7.4. alt_avalon_sgdma_do_sync_transfer()

Prototype:	alt_u8 alt_avalon_sgdma_do_sync_transfer(alt_sgdma_dev *dev, alt_sgdma_descriptor *desc)
Thread-safe:	No.
Available from ISR:	Not recommended.
Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs**.h>
Parameters:	*dev—a pointer to an SG-DMA device structure. *desc—a pointer to a single, constructed descriptor. The descriptor must have its "next" descriptor field initialized either to a non-ready descriptor, or to the next descriptor in the chain.
Returns:	Returns the contents of the status register.
Description:	Sends a fully formed descriptor or list of descriptors to the SG-DMA controller for transfer. This function blocks both before transfer, if the SG-DMA controller is busy, and until the requested transfer has completed. If an error is detected during the transfer, it is abandoned and the controller's status register contents are returned to the caller. Additional error information is available in the status bits of each descriptor that the SG-DMA processed. The user application searches through the descriptor or list of descriptors to gather specific error information. The run bit is cleared before the beginning of the transfer and is set to 1 to restart a new descriptor chain.

31.7.5. alt_avalon_sgdma_construct_mem_to_mem_desc()

Prototype:	void alt_avalon_sgdma_construct_mem_to_mem_desc(alt_sgdma_descriptor *desc, alt_sgdma_descriptor *next, alt_u32 *read_addr, alt_u32 *write_addr, alt_u16 length, int read_fixed, int write_fixed)
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs**.h>
Parameters:	*desc—a pointer to the descriptor being constructed. *next—a pointer to the "next" descriptor. This does not need to be a complete or functional descriptor, but must be properly allocated. *read_addr—the first read address for the SG-DMA transfer. *write_addr—the first write address for the SG-DMA transfer. length—the number of bytes for the transfer. read_fixed—if non-zero, the SG-DMA reads from a fixed address. write_fixed—if non-zero, the SG-DMA writes to a fixed address.
Returns:	void
Description:	This function constructs a single SG-DMA descriptor in the memory specified in `alt_avalon_sgdma_descriptor *desc` for an Avalon-MM to Avalon-MM transfer. The function sets the OWNED_BY_HW bit in the descriptor's control field, marking the completed descriptor as ready to run. The descriptor is processed when the SG-DMA controller receives the descriptor and the RUN bit is 1. The next field of the descriptor being constructed is set to the address in *next. The OWNED_BY_HW bit of the descriptor at *next is explicitly cleared. Once the SG-DMA completes processing of the *desc, it does not process the descriptor at *next until its OWNED_BY_HW bit is set. To create a descriptor chain, you can repeatedly call this function using the previous call's *next pointer in the *desc parameter. You must properly allocate memory for the creation of both the descriptor under construction as well as the next descriptor in the chain. Descriptors must be in a memory device hosted by the SG-DMA controller's chain read and chain write Avalon host ports. Care must be taken to ensure that both *desc and *next point to areas of memory hosted by the controller.

31.7.6. alt_avalon_sgdma_construct_stream_to_mem_desc()

Prototype:	void alt_avalon_sgdma_construct_stream_to_mem_desc(alt_sgdma_descriptor *desc, alt_sgdma_descriptor *next, alt_u32 *write_addr, alt_u16 length_or_eop, int write_fixed)
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs**.h>
Parameters:	*desc—a pointer to the descriptor being constructed. *next—a pointer to the "next" descriptor. This does not need to be a complete or functional descriptor, but must be properly allocated. *write_addr—the first write address for the SG-DMA transfer. length_or_eop—the number of bytes for the transfer. If set to zero (0x0), the transfer continues until an EOP signal is received from the Avalon-ST interface. write_fixed—if non-zero, the SG-DMA will write to a fixed address.
Returns:	void
Description:	This function constructs a single SG-DMA descriptor in the memory specified in `alt_avalon_sgdma_descriptor` *desc for an Avalon-ST to Avalon-MM transfer. The source (read) data for the transfer comes from the Avalon-ST interface connected to the SG-DMA controller's streaming read port. The function sets the OWNED_BY_HW bit in the descriptor's control field, marking the completed descriptor as ready to run. The descriptor is processed when the SG-DMA controller receives the descriptor and the RUN bit is 1. The next field of the descriptor being constructed is set to the address in *next. The OWNED_BY_HW bit of the descriptor at *next is explicitly cleared. Once the SG-DMA completes processing of the *desc, it does not process the descriptor at *next until its OWNED_BY_HW bit is set. To create a descriptor chain, you can repeatedly call this function using the previous call's *next pointer in the *desc parameter. You must properly allocate memory for the creation of both the descriptor under construction as well as the next descriptor in the chain. Descriptors must be in a memory device hosted by the SG-DMA controller's chain read and chain write Avalon host ports. Care must be taken to ensure that both *desc and *next point to areas of memory hosted by the controller.

31.7.7. alt_avalon_sgdma_construct_mem_to_stream_desc()

Prototype:	void alt_avalon_sgdma_construct_mem_to_stream_desc(alt_sgdma_descriptor *desc, alt_sgdma_descriptor *next, alt_u32 *read_addr, alt_u16 length, int read_fixed, int generate_sop, int generate_eop, alt_u8 atlantic_channel)
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs**.h>
Parameters:	*desc—a pointer to the descriptor being constructed. *next—a pointer to the "next" descriptor. This does not need to be a complete or functional descriptor, but must be properly allocated. *read_addr—the first read address for the SG-DMA transfer. length—the number of bytes for the transfer. read_fixed—if non-zero, the SG-DMA reads from a fixed address. generate_sop—if non-zero, the SG-DMA generates a SOP on the Avalon-ST interface when commencing the transfer. generate_eop—if non-zero, the SG-DMA generates an EOP on the Avalon-ST interface when completing the transfer.

continued...

	atlantic_channel—an 8-bit Avalon-ST channel number. Channels are currently not supported. Set this parameter to 0.
Returns:	void
Description:	This function constructs a single SG-DMA descriptor in the memory specified in `alt_avalon_sgdma_descriptor` *desc for an Avalon-MM to Avalon-ST transfer. The destination (write) data for the transfer goes to the Avalon-ST interface connected to the SG-DMA controller's streaming write port. The function sets the OWNED_BY_HW bit in the descriptor's control field, marking the completed descriptor as ready to run. The descriptor is processed when the SG-DMA controller receives the descriptor and the RUN bit is 1. The next field of the descriptor being constructed is set to the address in *next. The OWNED_BY_HW bit of the descriptor at *next is explicitly cleared. Once the SG-DMA completes processing of the *desc, it does not process the descriptor at *next until its OWNED_BY_HW bit is set. To create a descriptor chain, you can repeatedly call this function using the previous call's *next pointer in the *desc parameter. You are responsible for properly allocating memory for the creation of both the descriptor under construction as well as the next descriptor in the chain. Descriptors must be in a memory device hosted by the SG-DMA controller's chain read and chain write Avalon host ports. Care must be taken to ensure that both *desc and *next point to areas of memory hosted by the controller.

31.7.8. alt_avalon_sgdma_enable_desc_poll()

Prototype:	void alt_avalon_sgdma_enable_desc_poll(alt_sgdma_dev *dev, alt_u32 frequency)
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs**.h> *dev—a pointer to an SG-DMA device structure.
Parameters:	`frequency`—the frequency value to set. Only the lower 11-bit value of the frequency is written to the control register.
Returns:	void
Description:	Enables descriptor polling mode with a specific frequency. There is no effect if the hardware does not support this mode.

31.7.9. alt_avalon_sgdma_disable_desc_poll()

Prototype:	void alt_avalon_sgdma_disable_desc_poll(alt_sgdma_dev *dev)
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs**.h>
Parameters:	*dev—a pointer to an SG-DMA device structure.
Returns:	void
Description:	Disables descriptor polling mode.

31.7.10. alt_avalon_sgdma_check_descriptor_status()

Prototype:	int alt_avalon_sgdma_check_descriptor_status(alt_sgdma_descriptor *desc)
Thread-safe:	Yes.
Available from ISR:	Yes.
	continued...

Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs**.h>
Parameters:	*desc—a pointer to the constructed descriptor to examine.
Returns:	Returns 0 if the descriptor is error-free, not owned by hardware, or a previously requested transfer completed normally. Other return codes are defined in **errno.h**.
Description:	Checks a descriptor previously owned by hardware for any errors reported in a previous transfer. The routine reports: errors reported by the SG-DMA controller, the buffer in use.

31.7.11. alt_avalon_sgdma_register_callback()

Prototype:	void alt_avalon_sgdma_register_callback(alt_sgdma_dev *dev, alt_avalon_sgdma_callback callback, alt_u16 chain_control, void *context)
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs**.h>
Parameters:	*dev—a pointer to the SG-DMA device structure. callback—a pointer to the callback routine to execute at interrupt level. chain_control—the SG-DMA control register contents. *context—a pointer used to pass context-specific information to the ISR. context can point to any ISR-specific information.
Returns:	void
Description:	Associates a user-specific routine with the SG-DMA interrupt handler. If a callback is registered, all non-blocking transfers enables interrupts that causes the callback to be executed. The callback runs as part of the interrupt service routine, and care must be taken to follow the guidelines for acceptable interrupt service routine behavior as described in the Nios II Software Developer's Handbook. To disable callbacks after registering one, call this routine with 0x0 as the callback argument.

31.7.12. alt_avalon_sgdma_start()

Prototype:	void alt_avalon_sgdma_start(alt_sgdma_dev *dev)
Thread-safe:	No.
Available from ISR:	Yes.
Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs**.h>
Parameters:	*dev—a pointer to the SG-DMA device structure.
Returns:	void
Description:	Starts the DMA engine and processes the descriptor pointed to in the controller's next descriptor pointer and all subsequent descriptors in the chain. It is not necessary to call this function when do_sync or do_async is used.

31.7.13. alt_avalon_sgdma_stop()

Prototype:	void alt_avalon_sgdma_stop(alt_sgdma_dev *dev)
Thread-safe:	No.
Available from ISR:	Yes.

continued...

Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs**.h>
Parameters:	*dev—a pointer to the SG-DMA device structure.
Returns:	void
Description:	Stops the DMA engine following completion of the current buffer descriptor. It is not necessary to call this function when do_sync or do_async is used.

31.7.14. alt_avalon_sgdma_open()

Prototype:	alt_sgdma_dev* alt_avalon_sgdma_open(const char* name)
Thread-safe:	Yes.
Available from ISR:	No.
Include:	<**altera_avalon_sgdma.h**>, <**altera_avalon_sgdma_descriptor.h**>, <**altera_avalon_sgdma_regs**.h>
Parameters:	name—the name of the SG-DMA device to open.
Returns:	A pointer to the SG-DMA device structure associated with the supplied name, or NULL if no corresponding SG-DMA device structure was found.
Description:	Retrieves a pointer to a hardware SG-DMA device structure.

31.8. Scatter-Gather DMA Controller Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
October 2015	2015.10.30	Updated sections: • Register Maps: "Control Register Bit Map" table • SG-DMA API: "Function List" table Added sections: • alt_avalon_sgdma_enable_desc_poll() • alt_avalon_sgdma_disable_desc_poll()
July 2014	2014.07.24	Updated Register Maps table, included **version** register
December 2010	v10.1.0	Updated figure 19-4 and figure 19-5. Revised the bit description of IE_GLOBAL in table 19-7. Removed the "Device Support", "Instantiating the Core in SOPC Builder", and "Referenced Documents" sections.
July 2010	v10.0.0	No change from previous release.
November 2009	v9.1.0	Revised descriptions of register fields and bits. Added description to the memory-to-stream configurations. Added descriptions to alt_avalon_sgdma_do_sync_transfer() and alt_avalon_sgdma_do_async_transfer() API. Added a list on error signals implementation.
March 2009	v9.0.0	Added description of **Enable bursting on descriptor read host**.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. Added section DMA Descriptors in Functional Specifications

continued...

Date	Version	Changes
		Revised descriptions of register fields and bits. Reorganized sections Software Programming Model and Programming with SG-DMA Controller Core.
May 2008	v8.0.0	Added sections on burst transfers.

32. SDRAM Controller Core

Attention: The SDRAM Controller Core is part of a product obsolescence and support discontinuation schedule. For new design, Intel recommends that you use other IPs with equivalent functions. To see a list of available IPs, refer to the Intel FPGA IP Portfolio page.

32.1. Core Overview

The SDRAM controller core with Avalon interface provides an Avalon Memory-Mapped (Avalon-MM) interface to off-chip SDRAM. The SDRAM controller allows designers to create custom systems in an Intel FPGA device that connect easily to SDRAM chips. The SDRAM controller supports standard SDRAM as described in the PC100 specification.

SDRAM is commonly used in cost-sensitive applications requiring large amounts of volatile memory. While SDRAM is relatively inexpensive, control logic is required to perform refresh operations, open-row management, and other delays and command sequences. The SDRAM controller connects to one or more SDRAM chips, and handles all SDRAM protocol requirements. Internal to the device, the core presents an Avalon-MM agent port that appears as linear memory (flat address space) to Avalon-MM host peripherals.

The core can access SDRAM subsystems with various data widths (8, 16, 32, or 64 bits), various memory sizes, and multiple chip selects. The Avalon-MM interface is latency-aware, allowing read transfers to be pipelined. The core can optionally share its address and data buses with other off-chip Avalon-MM tri-state devices. This feature is valuable in systems that have limited I/O pins, yet must connect to multiple memory chips in addition to SDRAM.

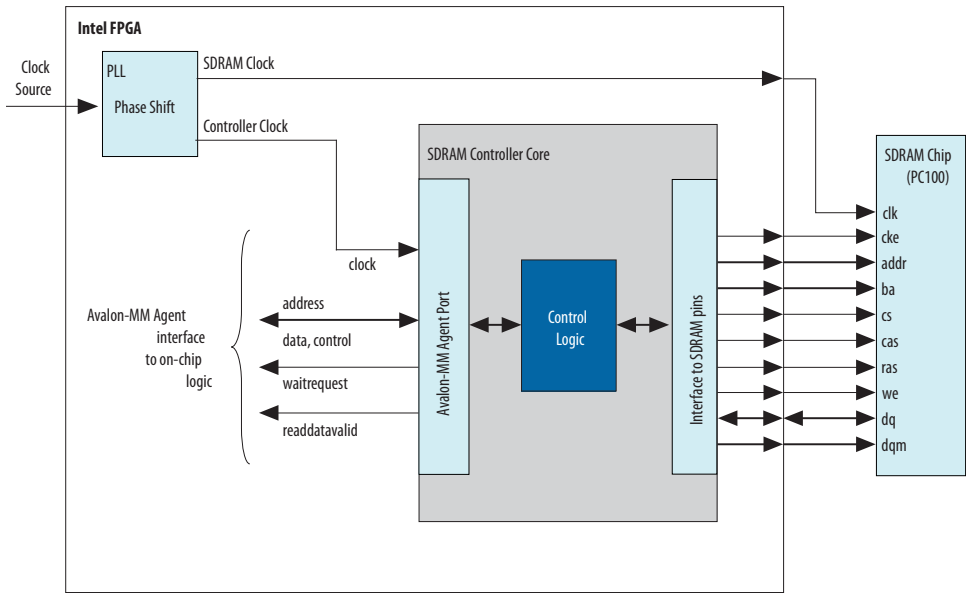
32.2. Functional Description

The diagram below shows a block diagram of the SDRAM controller core connected to an external SDRAM chip.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Figure 96. SDRAM Controller with Avalon Interface Block Diagram



The following sections describe the components of the SDRAM controller core in detail. All options are specified at system generation time, and cannot be changed at runtime.

Related Information

SDRAM Controller Core on page 405

32.2.1. Avalon-MM Interface

The Avalon-MM agent port is the user-visible part of the SDRAM controller core. The agent port presents a flat, contiguous memory space as large as the SDRAM chip(s). When accessing the agent port, the details of the PC100 SDRAM protocol are entirely transparent. The Avalon-MM interface behaves as a simple memory interface. There are no memory-mapped configuration registers.

The Avalon-MM agent port supports peripheral-controlled wait states for read and write transfers. The agent port stalls the transfer until it can present valid data. The agent port also supports read transfers with variable latency, enabling high-bandwidth, pipelined read transfers. When a host peripheral reads sequential addresses from the agent port, the first data returns after an initial period of latency. Subsequent reads can produce new data every clock cycle. However, data is not guaranteed to return every clock cycle, because the SDRAM controller must pause periodically to refresh the SDRAM.

For details about Avalon-MM transfer types, refer to the Avalon Interface Specifications.

32.2.2. Off-Chip SDRAM Interface

The interface to the external SDRAM chip presents the signals defined by the PC100 standard. These signals must be connected externally to the SDRAM chip(s) through I/O pins on the Intel FPGA device.

32.2.2.1. Signal Timing and Electrical Characteristics

The timing and sequencing of signals depends on the configuration of the core. The hardware designer configures the core to match the SDRAM chip chosen for the system. See the **Configuration** section for details. The electrical characteristics of the device pins depend on both the target device family and the assignments made in the Intel Quartus Prime software. Some device families support a wider range of electrical standards, and therefore are capable of interfacing with a greater variety of SDRAM chips. For details, refer to the device handbook for the target device family.

32.2.2.2. Synchronizing Clock and Data Signals

The clock for the SDRAM chip (SDRAM clock) must be driven at the same frequency as the clock for the Avalon-MM interface on the SDRAM controller (controller clock). As in all synchronous designs, you must ensure that address, data, and control signals at the SDRAM pins are stable when a clock edge arrives. As shown in the above **SDRAM Controller with Avalon Interface block diagram**, you can use an on-chip phase-locked loop (PLL) to alleviate clock skew between the SDRAM controller core and the SDRAM chip. At lower clock speeds, the PLL might not be necessary. At higher clock rates, a PLL is necessary to ensure that the SDRAM clock toggles only when signals are stable on the pins. The PLL block is not part of the SDRAM controller core. If a PLL is necessary, you must instantiate it manually. You can instantiate the PLL core interface or instantiate an ALTPLL IP core outside the Platform Designer system module.

If you use a PLL, you must tune the PLL to introduce a clock phase shift so that SDRAM clock edges arrive after synchronous signals have stabilized. See **Clock, PLL and Timing Considerations** sections for details.

For more information about instantiating a PLL, refer to **PLL Cores** chapter. The Nios II development tools provide example hardware designs that use the SDRAM controller core in conjunction with a PLL, which you can use as a reference for your custom designs.

The Nios II development tools are available free for download fromIntel FPGA website.

32.2.2.3. Clock Disable Mode Support

The SDRAM controller does not support clock-disable modes. The SDRAM controller permanently asserts the CKE signal on the SDRAM.

32.2.2.4. Sharing Pins with other Avalon-MM Tri-State Devices

If an Avalon-MM tri-state bridge is present, the SDRAM controller core can share pins with the existing tri-state bridge. In this case, the core's `addr`, `dq` (data) and `dqm` (byte-enable) pins are shared with other devices connected to the Avalon-MM tri-state bridge. This feature conserves I/O pins, which is valuable in systems that have multiple external memory chips (for example, flash, SRAM, and SDRAM), but too few pins to dedicate to the SDRAM chip. See **Performance Considerations** section for details about how pin sharing affects performance.

The SDRAM addresses must connect all address bits regardless of the size of the word so that the low-order address bits on the tri-state bridge align with the low-order address bits on the memory device. The Avalon-MM tristate address signal always

presents a byte address. It is not possible to drop A0 of the tri-state bridge for memories when the smallest access size is 16 bits or A0-A1 of the tri-state bridge when the smallest access size is 32 bits.

32.2.3. Board Layout and Pinout Considerations

When making decisions about the board layout and device pinout, try to minimize the skew between the SDRAM signals. For example, when assigning the device pinout, group the SDRAM signals, including the SDRAM clock output, physically close together. Also, you can use the **Fast Input Register** and **Fast Output Register** logic options in the Intel Quartus Prime software. These logic options place registers for the SDRAM signals in the I/O cells. Signals driven from registers in I/O cells have similar timing characteristics, such as t_{CO}, t_{SU}, and t_H.

32.2.4. Performance Considerations

Under optimal conditions, the SDRAM controller core's bandwidth approaches one word per clock cycle. However, because of the overhead associated with refreshing the SDRAM, it is impossible to reach one word per clock cycle. Other factors affect the core's performance, as described in the following sections.

32.2.4.1. Open Row Management

SDRAM chips are arranged as multiple banks of memory, in which each bank is capable of independent open-row address management. The SDRAM controller core takes advantage of open-row management for a single bank. Continuous reads or writes within the same row and bank operate at rates approaching one word per clock. Applications that frequently access different destination banks require extra management cycles to open and close rows.

32.2.4.2. Sharing Data and Address Pins

When the controller shares pins with other tri-state devices, average access time usually increases and bandwidth decreases. When access to the tri-state bridge is granted to other devices, the SDRAM incurs overhead to open and close rows. Furthermore, the SDRAM controller has to wait several clock cycles before it is granted access again.

To maximize bandwidth, the SDRAM controller automatically maintains control of the tri-state bridge as long as back-to-back read or write transactions continue within the same row and bank.

This behavior may degrade the average access time for other devices sharing the Avalon-MM tri-state bridge.

The SDRAM controller closes an open row whenever there is a break in back-to-back transactions, or whenever a refresh transaction is required. As a result:

- The controller cannot permanently block access to other devices sharing the tri-state bridge.
- The controller is guaranteed not to violate the SDRAM's row open time limit.

32.2.4.3. Hardware Design and Target Device

The target device affects the maximum achievable clock frequency of a hardware design. Certain device families achieve higher f_{MAX} performance than other families. Furthermore, within a device family, faster speed grades achieve higher performance. The SDRAM controller core can achieve 100 MHz in Intel FPGA high-performance device families, such as Stratix series. However, the core might not achieve 100 MHz performance in all Intel FPGA device families.

The f_{MAX} performance also depends on the system design. The SDRAM controller clock can also drive other logic in the system module, which might affect the maximum achievable frequency. For the SDRAM controller core to achieve f_{MAX} performance of 100 MHz, all components driven by the same clock must be designed for a 100 MHz clock rate, and timing analysis in the Intel Quartus Prime software must verify that the overall hardware design is capable of 100 MHz operation.

32.3. Configuration

The SDRAM controller MegaWizard has two pages: **Memory Profile** and **Timing**. This section describes the options available on each page.

The **Presets** list offers several pre-defined SDRAM configurations as a convenience. If the SDRAM subsystem on the target board matches one of the preset configurations, you can configure the SDRAM controller core easily by selecting the appropriate preset value. The following preset configurations are defined:

- Micron MT8LSDT1664HG module
- Four SDR100 8 MByte × 16 chips
- Single Micron MT48LC2M32B2-7 chip
- Single Micron MT48LC4M32B2-7 chip
- Single NEC D4564163-A80 chip (64 MByte × 16)
- Single Alliance AS4LC1M16S1-10 chip
- Single Alliance AS4LC2M8S0-10 chip

  Selecting a preset configuration automatically changes values on the **Memory Profile** and **Timing** tabs to match the specific configuration. Altering a configuration setting on any page changes the **Preset** value to **custom**.

32.3.1. Memory Profile Page

The **Memory Profile** page allows you to specify the structure of the SDRAM subsystem such as address and data bus widths, the number of chip select signals, and the number of banks.

Table 344. Memory Profile Page Settings

Settings		Allowed Values	Default Values	Description
Data Width		8, 16, 32, 64	32	SDRAM data bus width. This value determines the width of the dq bus (data) and the dqm bus (byte-enable).
Architecture Settings	Chip Selects	1, 2, 4, 8	1	Number of independent chip selects in the SDRAM subsystem. By using multiple chip selects, the SDRAM controller can combine multiple SDRAM chips into one memory subsystem.
	Banks	2, 4	4	Number of SDRAM banks. This value determines the width of the ba bus (bank address) that connects to the SDRAM. The correct value is provided in the data sheet for the target SDRAM.
Address Width Settings	Row	11, 12, 13, 14	12	Number of row address bits. This value determines the width of the addr bus. The Row and Column values depend on the geometry of the chosen SDRAM. For example, an SDRAM organized as 4096 (2^{12}) rows by 512 columns has a Row value of 12.
	Column	>= 8, and less than Row value	8	Number of column address bits. For example, the SDRAM organized as 4096 rows by 512 (2^{9}) columns has a Column value of 9.
Share pins via tri-state bridge dq/dqm/addr I/O pins		On, Off	Off	When set to No, all pins are dedicated to the SDRAM chip. When set to Yes, the addr, dq, and dqm pins can be shared with a tristate bridge in the system. In this case, select the appropriate tristate bridge from the pull-down menu.
Include a functional memory model in the system testbench		On, Off	On	When on, Platform Designer functional simulation model for the SDRAM chip. This default memory model accelerates the process of creating and verifying systems that use the SDRAM controller. See **Hardware Simulation Considerations** section.

Based on the settings entered on the **Memory Profile** page, the wizard displays the expected memory capacity of the SDRAM subsystem in units of megabytes, megabits, and number of addressable words. Compare these expected values to the actual size of the chosen SDRAM to verify that the settings are correct.

32.3.2. Timing Page

The **Timing** page allows designers to enter the timing specifications of the SDRAM chip(s) used. The correct values are available in the manufacturer's data sheet for the target SDRAM.

Table 345. Timing Page Settings

Settings	Allowed Values	Default Value	Description
CAS latency	1, 2, 3	3	Latency (in clock cycles) from a read command to data out.
Initialization refresh cycles	1–8	2	This value specifies how many refresh cycles the SDRAM controller performs as part of the initialization sequence after reset.
Issue one refresh command every	—	15.625 µs	This value specifies how often the SDRAM controller refreshes the SDRAM. A typical SDRAM requires 4,096 refresh commands every 64 ms, which can be achieved by issuing one refresh command every 64 ms / 4,096 = 15.625 µs.
Delay after power up, before initialization	—	100 µs	The delay from stable clock and power to SDRAM initialization.
Duration of refresh command (t_rfc)	—	70 ns	Auto Refresh period.
Duration of precharge command (t_rp)	—	20 ns	Precharge command period.
ACTIVE to READ or WRITE delay (t_rcd)	—	20 ns	ACTIVE to READ or WRITE delay.
Access time (t_ac)	—	17 ns	Access time from clock edge. This value may depend on CAS latency.
Write recovery time (t_wr, No auto precharge)	—	14 ns	Write recovery if explicit precharge commands are issued. This SDRAM controller always issues explicit precharge commands.

Regardless of the exact timing values you specify, the actual timing achieved for each parameter is an integer multiple of the Avalon clock period. For the **Issue one refresh command every** parameter, the actual timing is the greatest number of clock cycles that does not exceed the target value. For all other parameters, the actual timing is the smallest number of clock ticks that provides a value greater than or equal to the target value.

32.4. Hardware Simulation Considerations

This section discusses considerations for simulating systems with SDRAM. Three major components are required for simulation:

- A simulation model for the SDRAM controller.
- A simulation model for the SDRAM chip(s), also called the memory model.
- A simulation testbench that wires the memory model to the SDRAM controller pins.

  Some or all of these components are generated by Platform Designer at system generation time.

32.4.1. SDRAM Controller Simulation Model

The SDRAM controller design files generated by Platform Designer are suitable for both synthesis and simulation. Some simulation features are implemented in the HDL using "translate on/off" synthesis directives that make certain sections of HDL code invisible to the synthesis tool.

The simulation features are implemented primarily for easy simulation of Nios and Nios II processor systems using the ModelSim* simulator. The SDRAM controller simulation model is not ModelSim specific. However, minor changes may be required to make the model work with other simulators.

If you change the simulation directives to create a custom simulation flow, be aware that Platform Designer overwrites existing files during system generation. Take precautions to ensure your changes are not overwritten.

Refer to AN 351: Simulating Nios II Processor Designs for a demonstration of simulation of the SDRAM controller in the context of Nios II embedded processor systems.

32.4.2. SDRAM Memory Model

This section describes the two options for simulating a memory model of the SDRAM chip(s).

32.4.2.1. Using the Generic Memory Model

If the **Include a functional memory model the system testbench** option is enabled at system generation, Platform Designer generates an HDL simulation model for the SDRAM memory. In the auto-generated system testbench, Platform Designer automatically wires this memory model to the SDRAM controller pins.

Using the automatic memory model and testbench accelerates the process of creating and verifying systems that use the SDRAM controller. However, the memory model is a generic functional model that does not reflect the true timing or functionality of real SDRAM chips. The generic model is always structured as a single, monolithic block of memory. For example, even for a system that combines two SDRAM chips, the generic memory model is implemented as a single entity.

32.4.2.2. Using the SDRAM Manufacturer's Memory Model

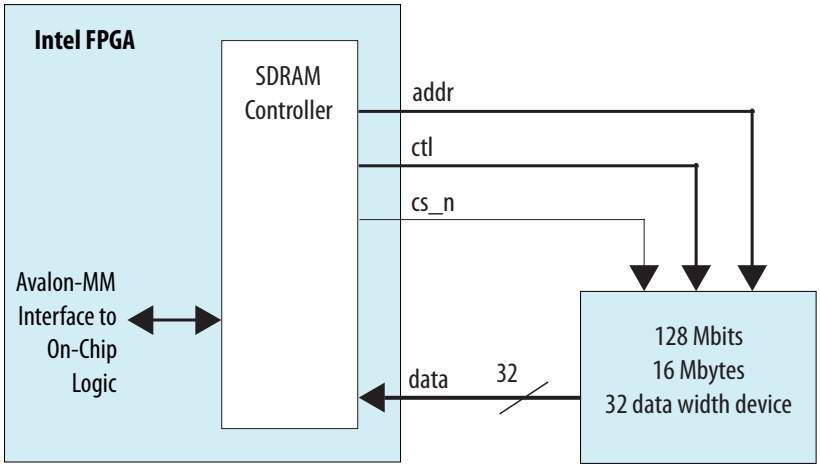
If the **Include a functional memory model the system testbench** option is not enabled, you are responsible for obtaining a memory model from the SDRAM manufacturer, and manually wiring the model to the SDRAM controller pins in the system testbench.

32.5. Example Configurations

The following examples show how to connect the SDRAM controller outputs to an SDRAM chip or chips. The bus labeled `ctl` is an aggregate of the remaining signals, such as `cas_n`, `ras_n`, `cke` and `we_n`.

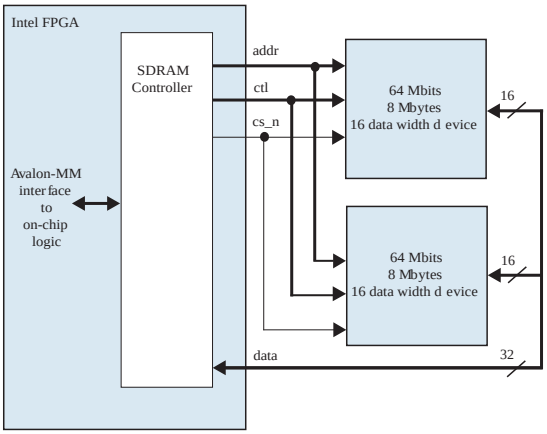
The address, data, and control signals are wired directly from the controller to the chip. The result is a 128-Mbit (16-Mbyte) memory space.

Figure 97. Single 128-Mbit SDRAM Chip with 32-Bit Data



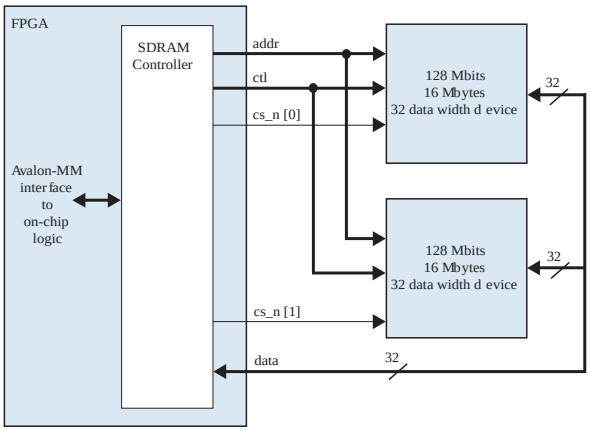
The address and control signals connect in parallel to both chips. The chips share the chipselect (cs_n) signal. Each chip provides half of the 32-bit data bus. The result is a logical 128-Mbit (16-Mbyte) 32-bit data memory.

Figure 98. Two 64-MBit SDRAM Chips Each with 16-Bit Data



The address, data, and control signals connect in parallel to the two chips. The chipselect bus (cs_n[1:0]) determines which chip is selected. The result is a logical 256-Mbit 32-bit wide memory.

Figure 99. Two 128-Mbit SDRAM Chips Each with 32-Bit Data



32.6. Software Programming Model

The SDRAM controller behaves like simple memory when accessed via the Avalon-MM interface. There are no software-configurable settings and no memory-mapped registers. No software driver routines are required for a processor to access the SDRAM controller.

32.7. Clock, PLL and Timing Considerations

This section describes issues related to synchronizing signals from the SDRAM controller core with the clock that drives the SDRAM chip. During SDRAM transactions, the address, data, and control signals are valid at the SDRAM pins for a window of time, during which the SDRAM clock must toggle to capture the correct values. At slower clock frequencies, the clock naturally falls within the valid window. At higher frequencies, you must compensate the SDRAM clock to align with the valid window.

Determine when the valid window occurs either by calculation or by analyzing the SDRAM pins with an oscilloscope. Then use a PLL to adjust the phase of the SDRAM clock so that edges occur in the middle of the valid window. Tuning the PLL might require trial-and-error effort to align the phase shift to the properties of your target board.

For details about the PLL circuitry in your target device, refer to the appropriate device family handbook.

For details about configuring the PLLs in Intel devices, refer to the ALTPLL IP Core User Guide.

32.7.1. Factors Affecting SDRAM Timing

The location and duration of the window depends on several factors:

- Timing parameters of the device and SDRAM I/O pins — I/O timing parameters vary based on device family and speed grade.
- Pin location on the device — I/O pins connected to row routing have different timing than pins connected to column routing.
- Logic options used during the Intel Quartus Prime compilation — Logic options such as the **Fast Input Register** and **Fast Output Register** logic affect the design fit. The location of logic and registers inside the device affects the propagation delays of signals to the I/O pins.
- SDRAM CAS latency

  As a result, the valid window timing is different for different combinations of FPGA and SDRAM devices. The window depends on the Intel Quartus Prime software fitting results and pin assignments.

32.7.2. Symptoms of an Untuned PLL

Detecting when the PLL is not tuned correctly might be difficult. Data transfers to or from the SDRAM might not fail universally. For example, individual transfers to the SDRAM controller might succeed, whereas burst transfers fail. For processor-based systems, if software can perform read or write data to SDRAM, but cannot run when the code is located in SDRAM, the PLL is probably tuned incorrectly.

32.7.3. Estimating the Valid Signal Window

This section describes how to estimate the location and duration of the valid signal window using timing parameters provided in the SDRAM datasheet and the Intel Quartus Prime software compilation report. After finding the window, tune the PLL so that SDRAM clock edges occur exactly in the middle of the window.

Calculating the window is a two-step process. First, determine by how much time the SDRAM clock can lag the controller clock, and then by how much time it can lead. After finding the maximum lag and lead values, calculate the midpoint between them.

These calculations provide an estimation only. The following delays can also affect proper PLL tuning, but are not accounted for by these calculations.

- Signal skew due to delays on the printed circuit board — These calculations assume zero skew.
- Delay from the PLL clock output nodes to destinations — These calculations assume that the delay from the PLL SDRAM-clock output-node to the pin is the same as the delay from the PLL controller-clock output-node to the clock inputs in the SDRAM controller. If these clock delays are significantly different, you must account for this phase shift in your window calculations.

  Lag is a negative time shift, relative to the controller clock, and lead is a positive time shift. The SDRAM clock can lag the controller clock by the lesser of the maximum lag for a read cycle or that for a write cycle. In other words, Maximum Lag = minimum(Read Lag, Write Lag). Similarly, the SDRAM clock can lead by the lesser of the maximum lead for a read cycle or for a write cycle. In other words, Maximum Lead = minimum(Read Lead, Write Lead).

Figure 100. Calculating the Maximum SDRAM Clock Lag

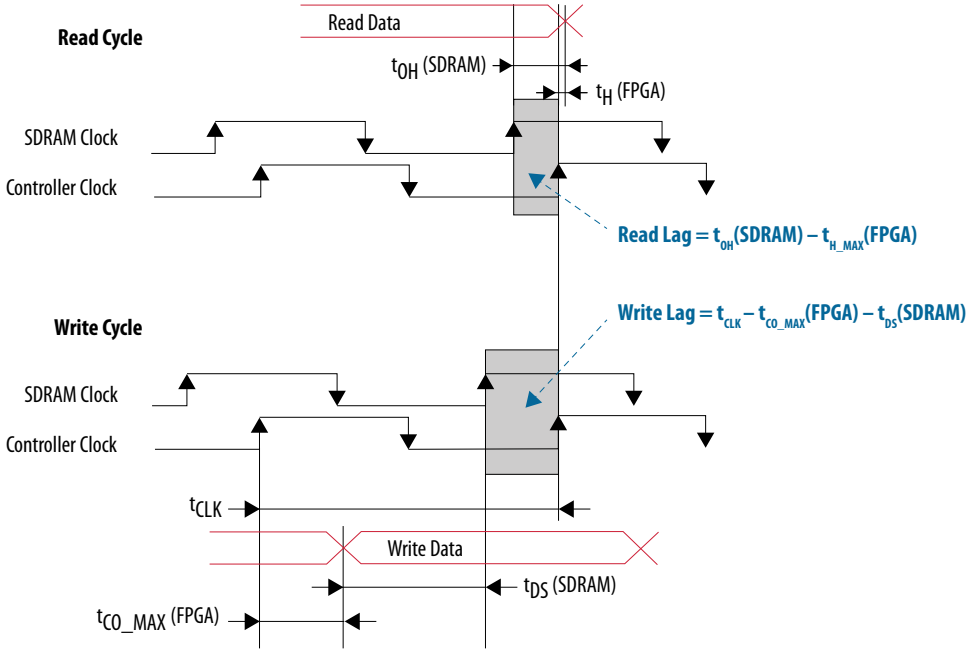
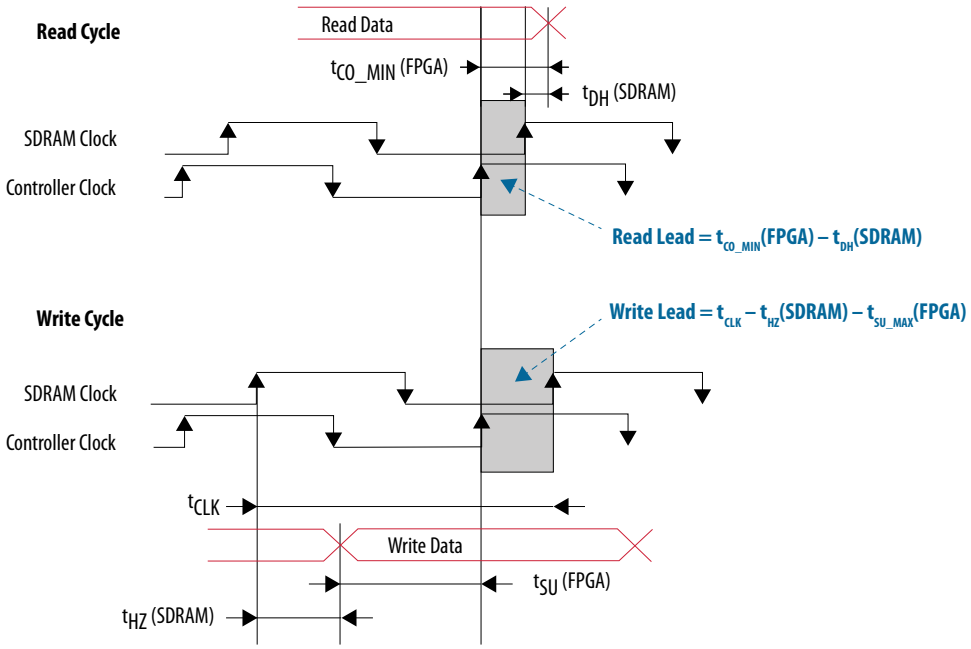


Figure 101. Calculating the Maximum SDRAM Clock Lead



32.7.4. Example Calculation

This section demonstrates a calculation of the signal window for a Micron MT48LC4M32B2-7 SDRAM chip and design targeting the Stratix II EP2S60F672C5 device. This example uses a CAS latency (CL) of 3 cycles, and a clock frequency of 50

MHz. All SDRAM signals on the device are registered in I/O cells, enabled with the **Fast Input Register** and **Fast Output Register** logic options in the Intel Quartus Prime software.

Table 346. Timing Parameters for Micron MT48LC4M32B2 SDRAM Device

Parameter		Symbol	Value (ns) in -7 Speed Grade	
			Min.	Max.
Access time from CLK (pos. edge)	CL = 3	$t_{AC(3)}$	—	5.5
	CL = 2	$t_{AC(2)}$	—	8
	CL = 1	$t_{AC(1)}$	—	17
Address hold time		t_{AH}	1	—
Address setup time		t_{AS}	2	—
CLK high-level width		t_{CH}	2.75	—
CLK low-level width		t_{CL}	2.75	—
Clock cycle time	CL = 3	$t_{CK(3)}$	7	—
	CL = 2	$t_{CK(2)}$	10	—
	CL = 1	$t_{CK(1)}$	20	—
CKE hold time		t_{CKH}	1	—
CKE setup time		t_{CKS}	2	—
CS#, RAS#, CAS#, WE#, DQM hold time		t_{CMH}	1	—
CS#, RAS#, CAS#, WE#, DQM setup time		t_{CMS}	2	—
Data-in hold time		t_{DH}	1	
Data-in setup time		t_{DS}	2	
Data-out high-impedance time	CL = 3	$t_{HZ(3)}$		5.5
	CL = 2	$t_{HZ(2)}$	—	8
	CL = 1	$t_{HZ(1)}$	—	17
Data-out low-impedance time		t_{LZ}	1	—
Data-out hold time		t_{OH}	2.5	

The FPGA I/O Timing Parameters table below shows the relevant timing information, obtained from the Timing Analyzer section of the Intel Quartus Prime Compilation Report. The values in the table are the maximum or minimum values among all device pins related to the SDRAM. The variance in timing between the SDRAM pins on the device is small (less than 100 ps) because the registers for these signals are placed in the I/O cell.

Table 347. FPGA I/O Timing Parameters

Parameter	Symbol	Value (ns)
Clock period	t_{CLK}	20
Minimum clock-to-output time	t_{CO_MIN}	2.399

continued...

Parameter	Symbol	Value (ns)
Maximum clock-to-output time	t_{CO_MAX}	2.477
Maximum hold time after clock	t_{H_MAX}	–5.607
Maximum setup time before clock	t_{SU_MAX}	5.936

You must compile the design in the Intel Quartus Prime software to obtain the I/O timing information for the design. Although Intel FPGA device family datasheets contain generic I/O timing information for each device, the Intel Quartus Prime Compilation Report provides the most precise timing information for your specific design.

The timing values found in the compilation report can change, depending on fitting, pin location, and other Intel Quartus Prime logic settings. When you recompile the design in the Intel Quartus Prime software, verify that the I/O timing has not changed significantly.

The following examples illustrate the calculations from figures Maximum SDRAM Clock Lag and Maximum Lead also using the values from the Timing Parameters and FPGA I/O Timing Parameters table.

The SDRAM clock can lag the controller clock by the lesser of Read Lag or Write Lag:

Read Lag = t_{OH}(SDRAM) – t_{H_MAX}(FPGA)

= 2.5 ns – (–5.607 ns) = 8.107 ns

or

Write Lag = t_{CLK} – t_{CO_MAX}(FPGA) – t_{DS}(SDRAM)

= 20 ns – 2.477 ns – 2 ns = 15.523 ns

The SDRAM clock can lead the controller clock by the lesser of Read Lead or Write Lead:

Read Lead = t_{CO_MIN}(FPGA) – t_{DH}(SDRAM)

= 2.399 ns – 1.0 ns = 1.399 ns

or

Write Lead = t_{CLK} – $t_{HZ(3)}$(SDRAM) – t_{SU_MAX}(FPGA)

= 20 ns – 5.5 ns – 5.936 ns = 8.564 ns

Therefore, for this example you can shift the phase of the SDRAM clock from –8.107 ns to 1.399 ns relative to the controller clock. Choosing a phase shift in the middle of this window results in the value (–8.107 + 1.399)/2 = –3.35 ns.

32.8. SDRAM Controller Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2021.03.29	21.1	Added product obsolescence and support discontinuation notice.
2019.12.16	19.4	Corrected the following figures: • *Calculating the Maximum SDRAM Clock Lag* • *Calculating the Maximum SDRAM Clock Lead*
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
May 2016	2016.05.03	Maintenance release.
July 2014	2014.07.24	Removed mention of SOPC Builder, updated to Platform Designer
December 2010	v10.1.0	Removed the "Device Support", "Instantiating the Core in SOPC Builder", and "Referenced Documents" sections.
July 2010	v10.0.0	No change from previous release.
November 2009	v9.1.0	No change from previous release.
March 2009	v9.0.0	No change from previous release.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	v8.0.0	No change from previous release.

33. Tri-State SDRAM Core

Attention: The Tri-State SDRAM Core is part of a product obsolescence and support discontinuation schedule. For new design, Intel recommends that you use other IPs with equivalent functions. To see a list of available IPs, refer to the Intel FPGA IP Portfolio page.

33.1. Core Overview

The Intel SDRAM Tri-State Controller core with Avalon interface provides an Avalon Memory-Mapped (Avalon-MM) interface to off-chip SDRAM. The SDRAM controller allows designers to create custom systems in an Intel FPGA device that connect easily to SDRAM chips. The SDRAM controller supports standard SDRAM defined by the PC100 specification.

SDRAM is commonly used in cost-sensitive applications requiring large amounts of volatile memory. While SDRAM is relatively inexpensive, control logic is required to perform refresh operations, open-row management, and other delays and command sequences. The SDRAM controller connects to one or more SDRAM chips, and handles all SDRAM protocol requirements. The SDRAM controller core presents an Avalon-MM agent port that appears as linear memory (flat address space) to Avalon-MM host peripherals.

The Avalon-MM interface is latency-aware, allowing read transfers to be pipelined. The core can optionally share its address and data buses with other off-chip Avalon-MM tri-state devices. This feature is valuable in systems that have limited I/O pins, yet must connect to multiple memory chips in addition to SDRAM.

The Intel SDRAM Tri-State Controller has the same functionality as the SDRAM Controller Core with the addition of the Tri-State feature.

Related Information

Avalon Interface Specifications

33.2. Feature Description

The Intel SDRAM Tri-State controller core has the following features:

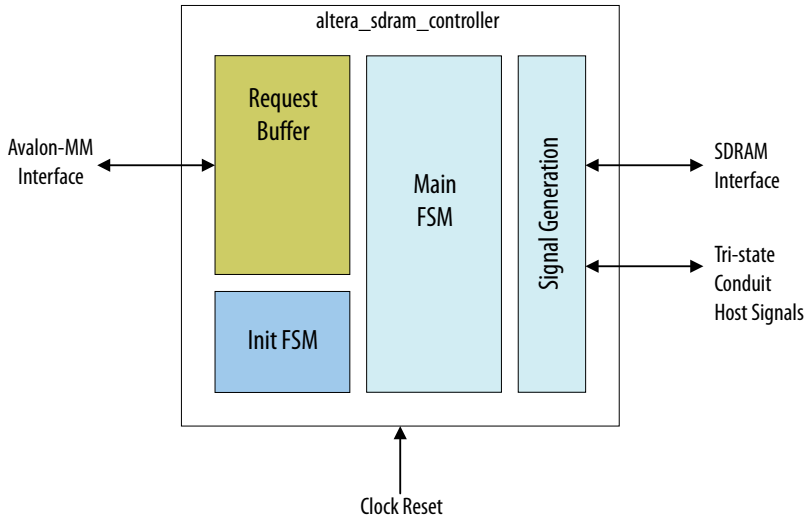
- Maximum frequency of 100-MHz
- Single clock domain design
- Sharing of `dq/dqm/addr I/`

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

33.2.1. Block Diagram

Figure 102. Tri-State SDRAM Block Diagram



33.3. Configuration Parameter

The following table shows the configuration parameters available for user to program during generation time of the IP core.

33.3.1. Memory Profile Page

The Memory Profile page allows you to specify the structure of the SDRAM subsystem such as address and data bus widths, the number of chip select signals, and the number of banks.

Table 348. Configuration Parameters

Parameter		GUI Legal Values	Default Values	Units
Data Width		8, 16, 32, 64	32	(Bit)s
Architecture	Chip Selects	1, 2, 4, 8	1	(Bit)s
	Banks	2, 4	4	(Bit)s
Address Widths	Row	11:14	12	(Bit)s
	Column	8:14	8	(Bit)s

33.3.2. Timing Page

The Timing page allows designers to enter the timing specifications of the Tri-State SDRAM chip(s) used. The correct values are available in the manufacturer's data sheet for the target SDRAM.

Table 349. Configuration Timing Parameters

Parameter	GUI Legal Values	Default Values	Units
CAS latency cycles	1, 2, 3	3	Cycles
Initialization refresh cycles	1:8	2	Cycles
Issue one refresh command every	0.0:156.25	15.625	us
Delay after power up, before initialization	0.0:999.0	100.00	us
Duration of refresh command (`t_rfc`)	0.0:700.0	70.0	ns
Duration of precharge command (`t_rp`)	0.0:200.0	20.0	ns
ACTIVE to READ or WRITE delay (`t_rcd`)	0.0:200.0	20.0	ns
Access time (`t_ac`)	0.0:999.0	5.5	ns
Write recovery time (`t_wr`, no auto precharge)	0.0:140.0	14.0	ns

33.4. Interface

The following are top level signals from core

Table 350. Clock and Reset Signals

Signal	Width	Direction	Description
`clk`	1	Input	System Clock
`rst_n`	1	Input	System asynchronous reset. The signal is asserted asynchronously, but is de-asserted synchronously after the rising edge of `ssi_clk`. The synchronization must be provided external to this component.

Table 351. Avalon-MM Agent Interface Signals

Signal	Width	Direction	Description
`avs_read`	1	Input	Avalon-MM read control. Asserted to indicate a read transfer. If present, `readdata` is required.
`avs_write`	1	Input	Avalon-MM write control. Asserted to indicate a write transfer. If present, `writedata` is required.
`avs_byteenable`	`dqm_width`	Input	Enables specific byte lane(s) during transfer. Each bit corresponds to a byte in `avs_writedata` and `avs_readdata`.
`avs_address`	`controller_addr_width`	Input	Avalon-MM address bus.
`avs_writedata`	`sdram_data_width`	Input	Avalon-MM write data bus. Driven by the bus host (bridge unit) during write cycles.

continued...

Signal	Width	Direction	Description
avs_readdata	sdram_data_width	Output	Avalon-MM readback data. Driven by the altera_spi during read cycles.
avs_readdatavalid	1	Output	Asserted to indicate that the avs_readdata signals contains valid data in response to a previous read request.
avs_waitrequest	1	Output	Asserted when it is unable to respond to a read or write request.

Table 352. Tristate Conduit Host / SDRAM Interface Signals

Signal	Width	Direction	Description
tcm_grant	1	Input	When asserted, indicates that a tristate conduit host has been granted access to perform transactions. tcm_grant is asserted in response to the tcm_request signal and remains asserted until 1 cycle following the deassertion of request. Valid only when pin sharing mode is enabled.
tcm_request	1	Output	The meaning of tcm_request depends on the state of the tcm_grant signal, as the following rules dictate: • When tcm_request is asserted and tcm_grant is deasserted, tcm_request is requesting access for the current cycle. • When tcm_request is asserted and tcm_grant is asserted, tcm_request is requesting access for the next cycle; consequently, tcm_request should be deasserted on the final cycle of an access. Because tcm_request is deasserted in the last cycle of a bus access, it can be reasserted immediately following the final cycle of a transfer, making both rearbitration and continuous bus access possible if no other hosts are requesting access. Once asserted, tcm_request must remain asserted until granted; consequently, the shortest bus access is 2 cycles. Valid only when pin-sharing mode is enabled.

continued...

Signal	Width	Direction	Description
sdram_dq_width	sdram_data_width	Output	SDRAM data bus output. Valid only when pin-sharing mode is enabled
sdram_dq_in	sdram_data_width	Input	SDRAM data bus input. Valid only when pin-sharing mode is enabled.
sdram_dq_oen	1	Output	SDRAM data bus output enable. Valid only when pin-sharing mode is enabled.
sdram_dq	sdram_data_width	Input/Output	SDRAM data bus. Valid only when pin-sharing mode is disabled.
sdram_addr	sdram_addr_width	Output	SDRAM address bus.
sdram_ba	sdram_bank_width	Output	SDRAM bank address.
sdram_dqm	dqm_width	Output	SDRAM data mask. When asserted, it indicates to the SDRAM chip that the corresponding data signal is suppressed. There is one DQM line per 8 bits data lines
sdram_ras_n	1	Output	Row Address Select. When taken LOW, the value on the tcm_addr_out bus is used to select the bank and activate the required row.
sdram_cas_n	1	Output	Column Address Select. When taken LOW, the value on the tcm_addr_out bus is used to select the bank and required column. A read or write operation will then be conducted from that memory location, depending on the state of tcm_we_out.
sdram_we_n	1	Output	SDRAM Write Enable, determinees whether the location addressed by tcm_addr_out is written to or read from. 0=Read 1=Write
sdram_cs_n		Output	SDRAM Chip Select. When taken LOW, will enables the SDRAM device.
sdram_cke	1	Output	SDRAM Clock Enable. The SDRAM controller does not support clock-disable modes. The SDRAM controller permanently asserts the tcm_sdr_cke_out signal on the SDRAM.

Note: The SDRAM controller does not have any configurable control status registers (CSR).

33.5. Reset and Clock Requirements

The main reset input signal to the SDRAM is treated as an asynchronous reset input from the SDRAM core perspective. A reset synchronizer circuit, as typically implemented for each reset domain in a complete SOC/ASIC system is not implemented within the SDRAM core. Instead, this reset synchronizer circuit should be implemented externally to the SDRAM, in a higher hierarchy within the complete system design, so that the "asynchronous assertion, synchronous de-assertion" rule is fulfilled.

The SDRAM core accepts an input clock at its `clk` input with maximum frequency of 100-MHz. The other requirements for the clock, such as its minimum frequency should be similar to the requirement of the external SDRAM which the SDRAM is interfaced to.

33.6. Architecture

The SDRAM Controller connects to one or more SDRAM chips, and handles all SDRAM protocol requirements. Internal to the device, the core presents an Avalon-MM agent ports that appears as a linear memory (flat address space) to Avalon-MM host device.

The core can access SDRAM subsystems with:

- Various data widths (8-, 16-, 32- or 64-bits)
- Various memory sizes
- Multiple chip selects

The Avalon-MM interface is latency-aware, allowing read transfers to be pipelined. The core can optionally share its address and data buses with other off-chip Avalon-MM tri-state devices.

Note: Limitations: for now the arbitration control of this mode should be handled by the host in the system to avoid a device monopolizing the shared buses.

Control logic within the SDRAM core responsible for the main functionality listed below, among others:

- Refresh operation
- Open_row management
- Delay and command management

Use of the data bus is intricate and thus requires a complex DRAM controller circuit. This is because data written to the DRAM must be presented in the same cycle as the write command, but reads produce output 2 or 3 cycles after the read command. The SDRAM controller must ensure that the data bus is never required for a read and a write at the same time.

33.6.1. Avalon-MM Agent Interface and CSR

The host processor perform data read and write operation to the external SDRAM devices through the Avalon-MM interface of the SDRAM core.

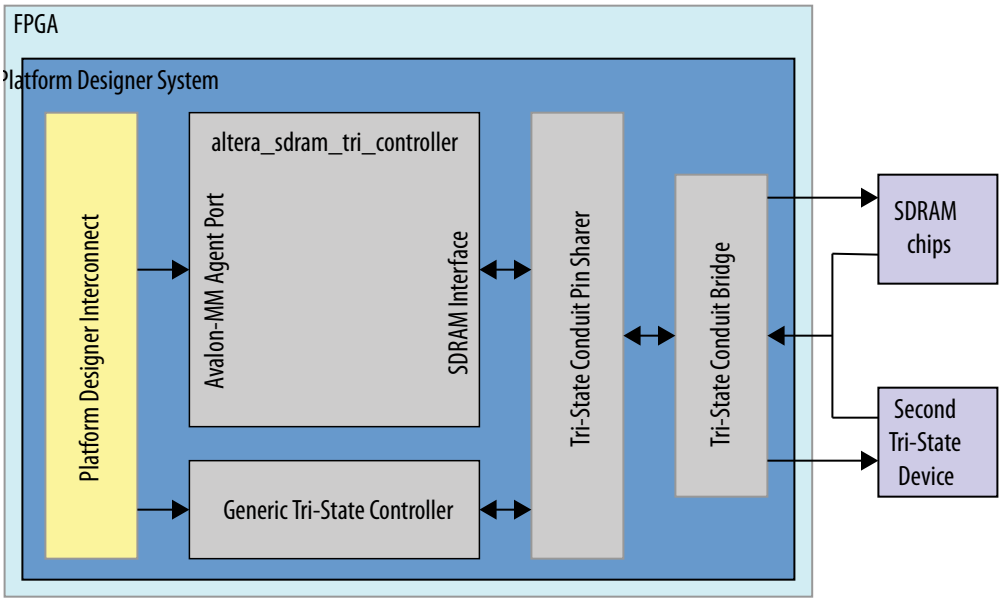
Please refer to *Avalon Interface Specifications* for more information on the details of the Avalon-MM Agent Interface.

Related Information

Avalon Interface Specifications

33.6.2. Block Level Usage Model

Figure 103. Shared-Bus System



33.7. Intel SDRAM Tri-State Controller Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2021.03.29	21.1	Added product obsolescence and support discontinuation notice.
2018.05.07	18.0	• Corrected the signal description for `sdram_dq_in` and `sdram_dq_oen`. • Updated *Figure: Shared-Bus System.*
2014.07.24	14.0	Initial release.

34. Video Sync Generator and Pixel Converter Cores

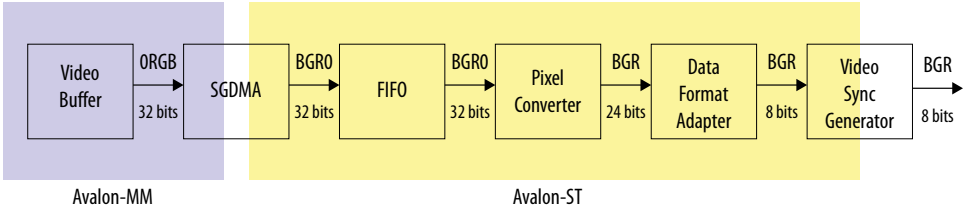
34.1. Core Overview

The video sync generator core accepts a continuous stream of pixel data in RGB format, and outputs the data to an off-chip display controller with proper timing. You can configure the video sync generator core to support different display resolutions and synchronization timings.

The pixel converter core transforms the pixel data to the format required by the video sync generator. The **Typical Placement in a System** figure shows a typical placement of the video sync generator and pixel converter cores in a system.

In this example, the video buffer stores the pixel data in 32-bit unpacked format. The extra byte in the pixel data is discarded by the pixel converter core before the data is serialized and sent to the video sync generator core.

Figure 104. Typical Placement in a System



These cores are deployed in the Nios II Embedded Software Evaluation Kit (NEEK), which includes an LCD display daughtercard assembly attached via an HSMC connector.

34.2. Video Sync Generator

This section describes the hardware structure and functionality of the video sync generator core.

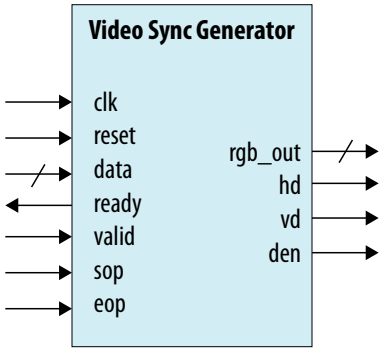
34.2.1. Functional Description

The video sync generator core adds horizontal and vertical synchronization signals to the pixel data that comes through its Avalon (Avalon-ST) input interface and outputs the data to an off-chip display controller. No processing or validation is performed on the pixel data.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Figure 105. Video Sync Generator Block Diagram



You can configure various aspects of the core and its Avalon-ST interface to suit your requirements. You can specify the data width, number of bits required to transfer each pixel and synchronization signals. See the **Parameters** section for more information on the available options.

To ensure incoming pixel data is sent to the display controller with correct timing, the video sync generator core must synchronize itself to the first pixel in a frame. The first active pixel is indicated by an sop pulse.

The video sync generator core expects continuous streams of pixel data at its input interface and assumes that each incoming packet contains the correct number of pixels (Number of rows * Number of columns). Data starvation disrupts synchronization and results in unexpected output on the display.

34.2.2. Parameters

Table 353. Video Sync Generator Parameters

Parameter Name	Description
Horizontal Sync Pulse Pixels	The width of the h-sync pulse in number of pixels.
Total Vertical Scan Lines	The total number of lines in one video frame. The value is the sum of the following parameters: **Number of Rows**, **Vertical Blank Lines**, and **Vertical Front Porch Lines**.
Number of Rows	The number of active scan lines in each video frame.
Horizontal Sync Pulse Polarity	The polarity of the h-sync pulse; 0 = active low and 1 = active high.
Horizontal Front Porch Pixels	The number of blanking pixels that follow the active pixels. During this period, there is no data flow from the Avalon-ST sink port to the LCD output data port.
Vertical Sync Pulse Polarity	The polarity of the v-sync pulse; 0 = active low and 1 = active high.
Vertical Sync Pulse Lines	The width of the v-sync pulse in number of lines.
Vertical Front Porch Lines	The number of blanking lines that follow the active lines. During this period, there is no data flow from the Avalon-ST sink port to the LCD output data port.
Number of Columns	The number of active pixels in each line.
Horizontal Blank Pixels	The number of blanking pixels that precede the active pixels. During this period, there is no data flow from the Avalon-ST sink port to the LCD output data port.
Total Horizontal Scan Pixels	The total number of pixels in one line. The value is the sum of the following parameters: **Number of Columns**, **Horizontal Blank Pixel**, and **Horizontal Front Porch Pixels**.

continued...

Parameter Name	Description
bits Per Pixel	The number of bits required to transfer one pixel. Valid values are 1 and 3. This parameter, when multiplied by **Data Stream Bit Width** must be equal to the total number of bits in one pixel. This parameter affects the operating clock frequency, as shown in the following equation: Operating clock frequency = (**bits per pixel**) * (Pixel_rate), where Pixel_rate (in MHz) = ((**Total Horizontal Scan Pixels**) * (**Total Vertical Scan Lines**) * (Display refresh rate in Hz))/1000000.
Vertical Blank Lines	The number of blanking lines that proceed the active lines. During this period, there is no data flow from the Avalon-ST sink port to the LCD output data port.
Data Stream Bit Width	The width of the inbound and outbound data.

34.2.3. Signals

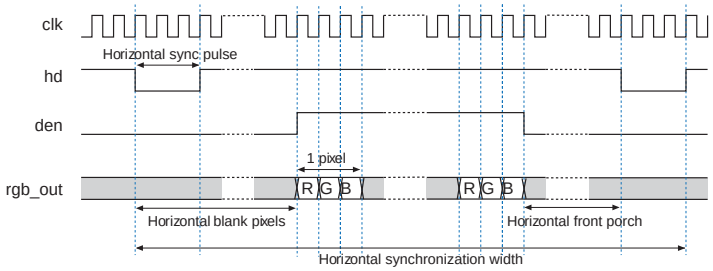
Table 354. Video Sync Generator Core Signals

Signal Name	Width (Bits)	Direction	Description
Global Signals			
`clk`	1	Input	System clock.
reset	1	Input	System reset.
Avalon-ST Signals			
`data`	Variable-width	Input	Incoming pixel data. The datawidth is determined by the parameter **Data Stream Bit Width**.
`ready`	1	Output	This signal is asserted when the video sync generator is ready to receive the pixel data.
`valid`	1	Input	This signal is not used by the video sync generator core because the core always expects valid pixel data on the next clock cycle after the `ready` signal is asserted.
`sop`	1	Input	Start-of-packet. This signal is asserted when the first pixel is received.
`eop`	1	Input	End-of-packet. This signal is asserted when the last pixel is received.
LCD Output Signals			
`rgb_out`	Variable-width	Output	Display data. The datawidth is determined by the parameter **Data Stream Bit Width**.
`hd`	1	Output	Horizontal synchronization pulse for display.
`vd`	1	Output	Vertical synchronization pulse for display.
`den`	1	Output	This signal is asserted when the video sync generator core outputs valid data for display.

34.2.4. Timing Diagrams

The horizontal and vertical synchronization timings are determined by the parameters setting. The table below shows the horizontal synchronization timing when the parameters **Data Stream Bit Width** and **bits Per Pixel** are set to 8 and 3, respectively.

Figure 106. Horizontal Synchronization Timing—8 Bits DataWidth and 3 bits Per Pixel



The table below shows the horizontal synchronization timing when the parameters **Data Stream Bit Width** and **bits Per Pixel** are set to 24 and 1, respectively.

Figure 107. Horizontal Synchronization Timing—24 Bits DataWidth and 1 Beat Per Pixel

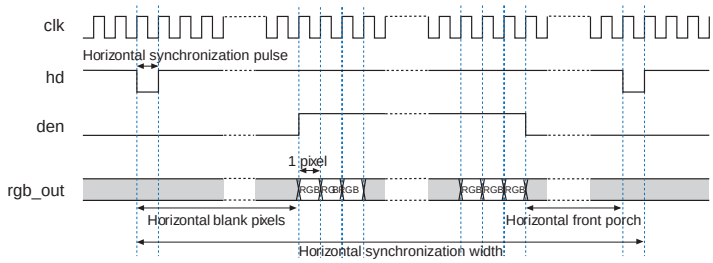
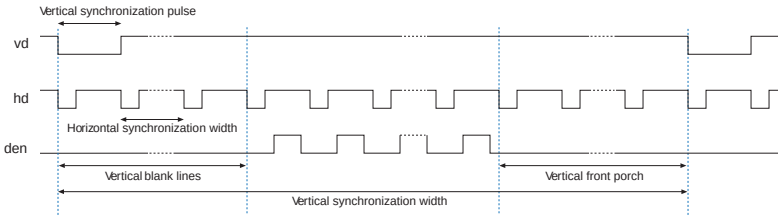


Figure 108. Vertical Synchronization Timing—8 Bits DataWidth and 3 bits Per Pixel / 24 Bits DataWidth and 1 Beat Per Pixel



34.3. Pixel Converter

This section describes the hardware structure and functionality of the pixel converter core.

34.3.1. Functional Description

The pixel converter core receives pixel data on its Avalon-ST input interface and transforms the pixel data to the format required by the video sync generator. The least significant byte of the 32-bit wide pixel data is removed and the remaining 24 bits are wired directly to the core's Avalon-ST output interface.

34.3.2. Parameters

You can configure the following parameter:

- **Source symbols per beat**—The number of symbols per beat on the Avalon-ST source interface.

34.3.3. Signals

Table 355. Pixel Converter Input Interface Signals

Signal Name	Width (Bits)	Direction	Description
Global Signals			
clk	1	Input	Not in use.
reset_n	1	Input	
Avalon-ST Signals			
data_in	32	Input	Incoming pixel data. Contains four 8-bit symbols that are transferred in 1 beat.
data_out	24	Output	Output data. Contains three 8-bit symbols that are transferred in 1 beat.
sop_in	1	Input	Wired directly to the corresponding output signals.
eop_in	1	Input	
ready_in	1	Input	
valid_in	1	Input	
empty_in	1	Input	
sop_out	1	Output	Wired directly from the input signals.
eop_out	1	Output	
ready_out	1	Output	
valid_out	1	Output	
empty_out	1	Output	

34.4. Hardware Simulation Considerations

For a typical 60 Hz refresh rate, set the simulation length for the video sync generator core to at least 16.7 µs to get a full video frame. Depending on the size of the video frame, simulation may take a very long time to complete.

34.5. Video Sync Generator and Pixel Converter Cores Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
July 2014	2014.07.24	Removed mention of SOPC Builder, updated to Platform Designer
December 2010	v10.1.0	Removed the “Device Support”, “Instantiating the Core in SOPC Builder”, and “Referenced Documents” sections.
July 2010	v10.0.0	No change from previous release.
November 2009	v9.1.0	No change from previous release.
March 2009	v9.0.0	No change from previous release.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	v8.0.0	Added new parameters for both cores.

35. Intel FPGA Interrupt Latency Counter Core

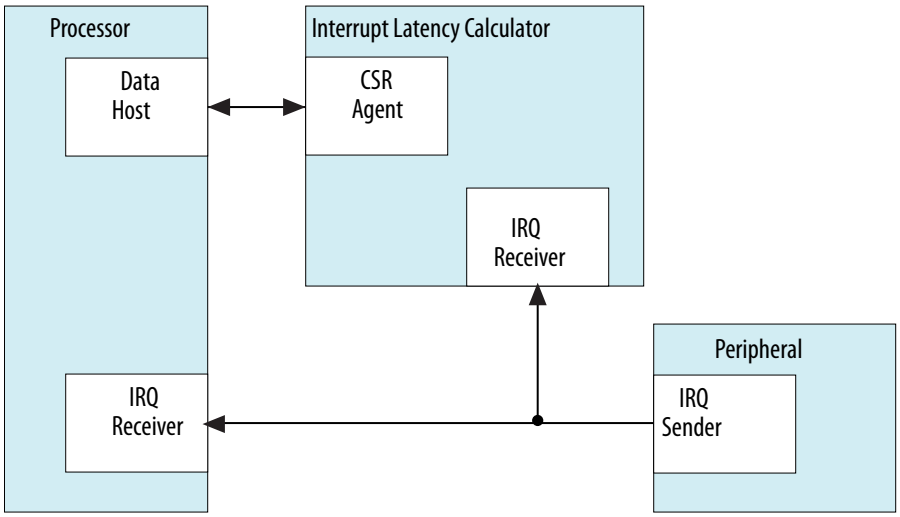
35.1. Core Overview

A processor running a program can be instructed to divert from its original execution path by an interrupt signal generated either by peripheral hardware or the firmware that is currently being executed. The processor now executes the portions of the program code that handles the interrupt requests known as Interrupt Service Routines (ISR) by moving to the instruction pointer to the ISR, and then continues operation. Upon completion of the routine, the processor returns to the previous location.

Intel FPGA's Interrupt Latency Calculator (ILC) is developed in mind to measure the time taken in terms of clock cycles to complete the interrupt service routine. Data obtained from the ILC is utilized by other latency sensitive IPs in order for it to maintain its proper operation. The data from the ILC can also be used to help the general firmware debugging exercise.

The ILC sits as a parallel to any interrupt receiver that will consume and perform an interrupt service routine. The following figure shows the orientation of a ILC in a system design.

Figure 109. Usage model of Interrupt Latency Calculator



35.2. Feature Description

The ILC is made up of three sub functional blocks. The top level interface is Avalon Memory Mapped (Avalon-MM) protocol compliant. The interrupt detector block will be activated by the rising edge of the interrupt signal or pulse, determined by a parameter during component generation. The interrupt detector block determines

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

when to start or stop the 32-bit internal counter, which is reset to zero every time it begins operation without affecting previous stored latency data register value. The latency data register is updated after the counter is stopped.

Each counter can be configured to host up to 32 identical counters to monitor separate IRQ channels. Each counter only observes one interrupt input. The interrupt could be level sensitive or pulse (edge) sensitive. In the case where more interrupt lines need to be monitored, multiple counters could be instantiated in Platform Designer.

ILC only keeps track of the latest interrupt latency value. If multiple interrupts are happening in series, only the last interrupt latency will be maintained. On the other hand, every start of interrupt edge refreshes the internal counter from zero.

35.2.1. Avalon-MM Compliant CSR Registers

Each ILC has rows of status registers each being 32 bits in length. The last five rows of CSR registers corresponding to address 0x20 to 0x24 are fixed regardless of the number of IRQ port count configured through the Platform Designer GUI. The Address 0x0 to 0x1F is reserved to store the latency value which depends on the number of IRQ port configured. For example, if you configure the instance to have only five counters, then only addressess 0x0 to 0x4 return a valid value when you try to read from it. When the IP user tries to read from an invalid address, the IP returns binary '0' value.

Table 356. ILC Register Mapping

Word Address Offset	Register/ Queue Name	Attribute
0x0	IRQ_0 Latency Data Registers	Read access only
0x1	IRQ_1 Latency Data Registers	Read access only
...	...	...
0x1F	IRQ_31 Latency Data Registers	Read access only
0x20	Control Registers	Read and Write access on LSB and Read only for the remaining bits
0x21	Frequency Registers	Read access only
0x22	Counter Stop Registers	Read and Write access
0x23	Read data Valid Registers	Read access only
0x24	IRQ Active Registers	Read access only

35.2.1.1. Control Register

Table 357. ILC Control Register Fields

Field Name	ILC Version		IRQ Port Count		IRQ TYPE	Global Enable
Bit Location	31	8	7	2	1	0

The control registers of the Interrupt Latency Counter is divided into four fields. The LSB is the global enable bit which by default stores a binary '0'. To enable the IP to work, it must be set to binary '1'. The next bit denotes the IRQ type the IP is configured to measure, with binary '0' indicating it is sensitive to level type IRQ signal;

while binary '1' means the IP is accepting pulse type interrupt signal. The next six bits stores the number of IRQ port count configured through the Platform Designer GUI. Bit 8 through bit 31 stores the revision value of the ILC instance.

35.2.1.2. Frequency Register

Table 358. Frequency Register

Field Name	System Frequency	
Bit Location	31	0

The frequency registers stores the clock frequency supplied to the IP. This 32-bit read only register holds system frequency data in Hz. For example, a 50 MHz clock signal is represented by hexadecimal 0x2FAF080.

35.2.1.3. Counter Stop Registers

Table 359. Counter Stop Registers

Field Name	Counter Stop Registers	
Bit Location	31	0

If the ILC is configured to support the pulse IRQ signal, then the counter stop registers are utilized by running software to halt the counter. Each bit corresponds to the IRQ port. For example, bit 0 controls `IRQ_0` counter. To stop the counter you have to write a binary '1' into the register. Counter stop registers do not affect the operation of the ILC in level mode.

Note: You need to clear the counter stop register to properly capture the next round of IRQ delay.

35.2.1.4. Latency Data Registers

Table 360. Latency Data Registers

Field Name	Latency Data Registers	
Bit Location	31	0

The latency data registers hold the latency value in terms of clock cycle from the moment the interrupt signal is fired until the IRQ signal goes low for level configuration or counter stop register being set for pulse configuration. This is a 32-bit read only register with each address corresponding to one IRQ port. The latency data registers can only be read three clock cycles after the IRQ signal goes low or when the counter stop registers are set to high in the level and pulse operating mode, respectively.

35.2.1.5. Data Valid Registers

Table 361. Data Valid Registers

Field Name	Data Valid Registers	
Bit Location	31	0

The data valid registers indicate whether the data from the latency data registers are ready to be read or not. By default, these registers hold a binary value of '0' out of reset. Once the counter data is transfered to the latency data register, the corresponding bit within the data valid register is set to binary '1'. It reverts back to binary '0' after a read operation has been consumed by the ILC.

35.2.1.6. IRQ Active Registers

Table 362. IRQ Active Registers

Field Name	IRQ Active Registers	
Bit Location	31	0

The IRQ active registers allow user to know which channel's counter is actively running. Each bit corresponds to the IRQ port. For example, bit 0 indicates the status of `IRQ_0`. If the IP is configured to support level IRQ, a logical high on the register indicates that the IRQ signal is held at logical 1. When the IP is configured to support pulse IRQ signal, a logical high on the register indicates that the pulse IRQ has been consumed and a logical 1 has not been written into the counter stop register.

35.2.2. 32-bit Counter

The 32-bit positive edge triggered D-flop base up counter takes in a reset signal which clears all the registers to zero. It also has an enable signal that determines when the counter operation is turned on or off.

35.2.3. Interrupt Detector

The interrupt detector can be customized to detect either signal edges or pulse using the Platform Designer interface. The interrupt detector generates an enable signal to start and stop the 32-bit counter.

35.3. Component Interface

Intel FPGA Interrupt Latency Calculator has an Avalon-MM agent interface which communicates with the Interrupt service routine initiator.

The table below shows the component interface that is available on the Intel FPGA Interrupt Latency Counter IP.

Table 363. Available Component Interfaces

Interface Port	Description	Remarks
Avalon-MM Agent (address , write, waitrequest , writedata[31:0], read, readdata[31:0])	Avalon-MM Agent interface for processor to talk to the IP.	This Avalon-MM Agent interface observes zero cycles read latency with waitrequest signal. The waitrequest signal defaults to binary '1' if there is no ongoing operation. If the Avalon-

continued...

Interface Port	Description	Remarks
		MM Read or Write signal goes high, the waitrequest signal only goes low if the readdata_valid_register goes high.
Clock	Clock input of component.	Clock signal to feed the latency counter logics.
Reset_n	Active LOW reset input/s.	Support asynchronous reset assertion. De-assertion of reset has to be synchronized to the input clock.
IRQ	IRQ signal from the interrupt signal initiator	Interrupt assertion and deassertion is synchronized to input clock.

35.4. Component Parameterization

The table below shows the configuration parameters available on the Intel FPGA Interrupt Latency Counter IP.

Table 364. Available Component Parameterizations

Parameter Name	Description	Default Value	Allowable Range
INTERRUPT_TYPE	**Value 0:** level sensitive interrupt input **Value 1:** edge/pulse interrupt input	0	0,1
CLOCK_RATE	Frequency of the clock signal that is connected to the IP	0	0 – 2^32
IRQ_PORT_COUNT	Configure number of IRQ PORT to use	32	1 - 32

35.5. Software Access

Since the component supports two types of incoming interrupts - level and edge/pulse, the software access routine for supporting each of the interrupt types has slightly different expectations.

35.5.1. Routine for Level Sensitive Interrupts

The software access routine for level sensitive interrupts is as follows:

1. Upon completion of ISR, read the data valid bit to ensure that the data is "valid" before reading the interrupt latency counter.
2. Read from the Latency Data Register to obtain the actual cycle spend for the interrupt.
   The value presented is in the amount of clock cycle associated with the clock connected to Interrupt Latency Counter.

35.5.2. Routine for Edge/Pulse Sensitive Interrupts

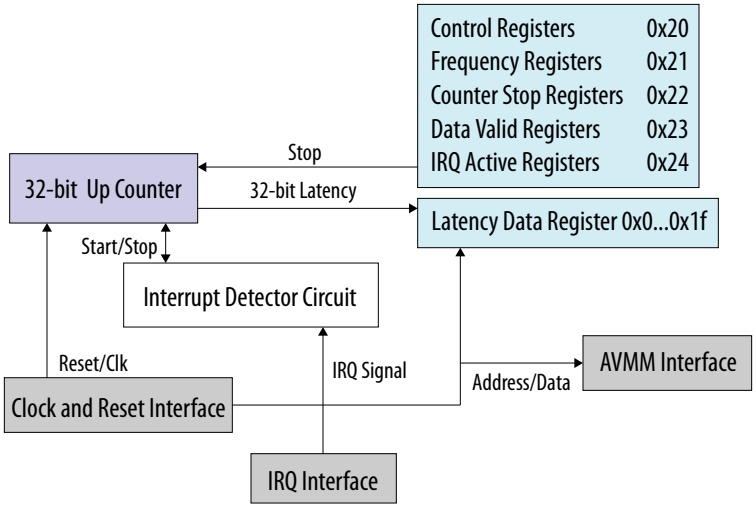
The software access routine for edge/pulse sensitive interrupts is as follows:

1. Upon completion of ISR, or at the end of ISR, software needs to write binary '1' to one of the 32-bit registers of the Counter Stop Register to stop the internal counter from counting. The LSB represents counter 0 and the MSB represents counter 31. This is the same as the level sensitive interrupt. Data valid bit is recommended to be read before reading the latency counter.
2. Read from Latency Data Register to obtain the actual cycle spend for the interrupt. The counter stop bit only needs clearing when the IP is configured to accept pulse IRQ. If level IRQ is employed. The counter stop bit is ignored.

35.6. Implementation Details

35.6.1. Interrupt Latency Counter Architecture

Figure 110. Interrupt Latency Calculator Architecture



The interrupt latency calculator operates on a single clock domain which is determined by which clock it is receiving at the CLK interface. The interrupt detector circuit is made up of a positive-edge triggered flop which delays the IRQ signal to be XORed with the original signal. The pulse resulted from the previous operation is then fed to an enable register where it will switch its state from logic 'low' to 'high'. This will trigger the counter to start its operation. Prior to this, the reset signal is assumed to be triggered through the firmware. Once the Interrupt service routine has been completed, the IRQ signal drops to logic low. This causes another pulse to be generated to stop the counter. Data from the counter is then duplicated into the latency data register to be read out.

When the interrupt detector is configured to react to a pulse signal, the incoming pulse is fed directly to enable the register to turn on the counter. In this mode, to halt the counter's operation, you have to write a Boolean '1' to the counter stop bit. Only the first IRQ pulse can trigger the counter to start counting and that subsequent pulse will not cause the counter to reset until a Boolean '1' is written into the counter stop register. In 'pulse' mode, the latency measured by the IP is one clock cycle more than actual latency.

35.7. IP Caveats

There are limitations in the Intel FPGA interrupt latency which the user needs to be aware of. This limitation arises due to the nature of state machines which incurs a period of clock cycle for state transitions.

1. The data latency registers cannot be read before a first IRQ is fired in any of the 32 channels. This causes the **Waitrequest** signal to be perpetually high which would lead to a system stall.
2. The data registers can only be read three clock cycles after the counter registers stop counting. These three clock cycles originate from the state machine moving from the **start** state to the **stop/store** state. It takes an additional clock cycle to propagate the data from the counter registers to the data store registers.
3. In the pulse IRQ mode, there is an idle cycle present between two consecutive write commands into the counter stop register. So, in the event that channel 1 is halted immediately after channel 0 is halted, then the minimum difference you see in the registered values is 2.
4. The interrupt latency counter will not notify you if an overflow occurs but the counter can count up to very huge numbers before an overflow happens. The magnitude of the delay numbers reported will suggest that the system has hung indefinitely.

35.8. Intel FPGA Interrupt Latency Counter Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0	• Clarified the information in *Avalon Memory-Mapped Interface Compliant CSR Registers* section. • Added a new section *IRQ Active Registers*. • Corrected the *Figure: Interrupt Latency Calculator Architecture*.

Date	Version	Changes
June 2016	2016.06.17	Updated: • Table 356 on page 434 Added word address offset 0x24 • Data Valid Registers on page 435 Updated description • Table 364 on page 437 Parameter name change • IP Caveats on page 439 Added limitation 4
July 2014	2014.07.24	Initial Release

36. Performance Counter Unit Core

36.1. Core Overview

The performance counter core with Avalon interface enables relatively unobtrusive, real-time profiling of software programs. With the performance counter, you can accurately measure execution time taken by multiple sections of code. You need only add a single instruction at the beginning and end of each section to be measured.

The main benefit of using the performance counter core is the accuracy of the profiling results. Alternatives include the following approaches:

- GNU profiler, `gprof`—`gprof` provides broad low-precision timing information about the entire software system. It uses a substantial amount of RAM, and degrades the real-time performance. For many embedded applications, `gprof` distorts real-time behavior too much to be useful.
- Interval timer peripheral—The interval timer is less intrusive than `gprof`. It can provide good results for narrowly targeted sections of code.

  The performance counter core is unobtrusive, requiring only a single instruction to start and stop profiling, and no RAM. It is appropriate for high-precision measurements of narrowly targeted sections of code.

  For further discussion of all three profiling methods, refer to AN 391: Profiling Nios II Systems.

  The core is designed for use in Avalon-based processor systems, such as a Nios II processor system. Intel FPGA device drivers enable the Nios II processor to use the performance counters.

36.2. Functional Description

The performance counter core is a set of counters which track clock cycles, timing multiple sections of your software. You can start and stop these counters in your software, individually or as a group. You can read cycle counts from hardware registers.

The core contains two counters for every section:

- Time: A 64-bit clock cycle counter.
- Events: A 32-bit event counter.

36.2.1. Section Counters

Each 64-bit time counter records the aggregate number of clock cycles spent in a section of code. The 32-bit event counter records the number of times the section executes.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

The performance counter core can have up to seven section counters.

36.2.2. Global Counter

The global counter controls all section counters. The section counters are enabled only when the global counter is running.

The 64-bit global clock cycle counter tracks the aggregate time for which the counters were enabled. The 32-bit global event counter tracks the number of global events, that is, the number of times the performance counter core has been enabled.

36.2.3. Register Map

The performance counter core has an Avalon Memory-Mapped (Avalon-MM) agent interface that provides access to memory-mapped registers. Reading from the registers retrieves the current times and event counts. Writing to the registers starts, stops, and resets the counters.

Table 365. Performance Counter Core Register Map

Offset	Register Name	Bit Description		
		Read	Write	
		31 ... 0	31 ... 1	0
0	$T[0]_{lo}$	global clock cycle counter [31: 0]	(1)	0 = STOP 1 = RESET
1	$T[0]_{hi}$	global clock cycle counter [63:32]	(1)	0 = START
2	Ev[0]	global event counter	(1)	(1)
3	—	(1)	(1)	(1)
4	$T[1]_{lo}$	section 1 clock cycle counter [31:0]	(1)	1 = STOP
5	$T[1]_{hi}$	section 1 clock cycle counter [63:32]	(1)	0 = START
6	Ev[1]	section 1 event counter	(1)	(1)
7	—	(1)	(1)	(1)
8	$T[2]_{lo}$	section 2 clock cycle counter [31:0]	(1)	1 = STOP
9	$T[2]_{hi}$	section 2 clock cycle counter [63:32]	(1)	0 = START
10	Ev[2]	section 2 event counter	(1)	(1)
11	—	(1)	(1)	(1)
. . .	. . .	. . .	. . .	. . .
4n + 0	$T[n]_{lo}$	section n clock cycle counter [31:0]	(1)	1 = STOP
4n + 1	$T[n]_{hi}$	section n clock cycle counter [63:32]	(1)	0 = START
4n + 2	Ev[n]	section n event counter	(1)	(1)
4n + 3	—	(1)	(1)	(1)

Note :
1. Reserved. Read values are undefined. When writing, set reserved bits to zero.

36.2.4. System Reset

After a system reset, the performance counter core is stopped and disabled, and all counters are set to zero.

36.3. Configuration

The following sections list the available options in the MegaWizard™ interface.

36.3.1. Define Counters

Choose the number of section counters you want to generate by selecting from the **Number of simultaneously-measured sections** list. The performance counter core may have up to seven sections. If you require more that seven sections, you can instantiate multiple performance counter cores.

36.3.2. Multiple Clock Domain Considerations

If your Platform Designer system uses multiple clocks, place the performance counter core in the same clock domain as the CPU. Otherwise, it is not possible to convert cycle counts to seconds correctly.

36.4. Hardware Simulation Considerations

You can use this core in simulation with no special considerations.

36.5. Software Programming Model

The following sections describe the software programming model for the performance counter core.

36.5.1. Software Files

Intel provides the following software files for Nios II systems. These files define the low-level access to the hardware and provide control and reporting functions. Do not modify these files.

- `altera_avalon_performance_counter.h`, `altera_avalon_performance_counter.c`—The header and source code for the functions and macros needed to control the performance counter core and retrieve raw results.
- `perf_print_formatted_report.c`—The source code for simple profile reporting.

36.5.2. Using the Performance Counter

In a Nios II system, you can control the performance counter core with a set of highly efficient C macros, and extract the results with C functions.

API Summary

The Nios II application program interface (API) for the performance counter core consists of functions, macros and constants.

Table 366. Performance Counter Macros and Functions

Name	Summary
`PERF_RESET()`	Stops and disables all counters, resetting them to 0.
`PERF_START_MEASURING()`	Starts the global counter and enables section counters.
`PERF_STOP_MEASURING()`	Stops the global counter and disables section counters.
`PERF_BEGIN()`	Starts timing a code section.
`PERF_END()`	Stops timing a code section.
`perf_print_formatted_report()`	Sends a formatted summary of the profiling results to `stdout`.
`perf_get_total_time()`	Returns the aggregate global profiling time in clock cycles.
`perf_get_section_time()`	Returns the aggregate time for one section in clock cycles.
`perf_get_num_starts()`	Returns the number of counter events.
`alt_get_cpu_freq()`	Returns the CPU frequency in Hz.

For a complete description of each macro and function, see the **Performance counter API** section.

Hardware Constants

You can get the performance counter hardware parameters from constants defined in `system.h`. The constant names are based on the performance counter instance name, specified on the **System Contents** tab in Platform Designer.

Table 367. Performance Counter Constants

Name (1)	Meaning
PERFORMANCE_COUNTER_BASE	Base address of core
PERFORMANCE_COUNTER_SPAN	Number of hardware registers
PERFORMANCE_COUNTER_HOW_MANY_SECTIONS	Number of section counters

Note :
1. Example based on instance name `performance_counter`.

Startup

Before using the performance counter core, invoke PERF_RESET to stop, disable and zero all counters.

Global Counter Usage

Use the global counter to enable and disable the entire performance counter core. For example, you might choose to leave profiling disabled until your software has completed its initialization.

Section Counter Usage

To measure a section in your code, surround it with the macros `PERF_BEGIN()` and `PERF_END()`. These macros consist of a single write to the performance counter core.

You can simultaneously measure as many code sections as you like, up to the number specified in Platform Designer. See the **Define Counters** section for details. You can start and stop counters individually, or as a group.

Typically, you assign one counter to each section of code you intend to profile. However, in some situations you may wish to group several sections of code in a single section counter. As an example, to measure general interrupt overhead, you can measure all interrupt service routines (ISRs) with one counter.

To avoid confusion, assign a mnemonic symbol for each section number.

Viewing Counter Values

Library routines allow you to retrieve and analyze the results. Use `perf_print_formatted_report()` to list the results to `stdout`, as shown below.

Table 368. Example 1:

```
perf_print_formatted_report(
     (void *)PERFORMANCE_COUNTER_BASE, // Peripheral's HW base address
     alt_get_cpu_freq(),               // defined in "system.h"
     3,                                // How many sections to print
     "1st checksum_test",              // Display-names of sections
     "pc_overhead",
     "ts_overhead");
```

The example below creates a table similar to this result.

Table 369. Example 2:

```
--Performance Counter Report--
Total Time: 2.07711 seconds (103855534 clock-cycles)
+-----------------+--------+-----------+---------------+-----------+
| Section         |    %   | Time (sec)| Time (clocks) |Occurrences|
+-----------------+--------+-----------+---------------+-----------+
|1st checksum_test|     50 |   1.03800 |      51899750 |         1 |
+-----------------+--------+-----------+---------------+-----------+
| pc_overhead     |1.73e-05|   0.00000 |            18 |         1 |
+-----------------+--------+-----------+---------------+-----------+
| ts_overhead     |4.24e-05|   0.00000 |            44 |         1 |
+-----------------+--------+-----------+---------------+-----------+
```

For full documentation of `perf_print_formatted_report()`, see the **Performance and Counter API** section.

36.5.3. Interrupt Behavior

The performance counter core does not generate interrupts.

You can start and stop performance counters, and read raw performance results, in an interrupt service routine (ISR). Do not call the `perf_print_formatted_report()` function from an ISR.

If an interrupt occurs during the measurement of a section of code, the time taken by the CPU to process the interrupt and return to the section is added to the measurement time. The same applies to context switches in a multi-threaded environment. Your software must take appropriate measures to avoid or handle these situations.

36.6. Performance Counter API

This section describes the application programming interface (API) for the performance counter core.

For Nios II processor users, Intel provides routines to access the performance counter core hardware. These functions are specific to the performance counter core and directly manipulate low level hardware. The performance counter core cannot be accessed via the HAL API or the ANSI C standard library.

36.6.1. PERF_RESET()

Prototype:	`PERF_RESET(p)`
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	**<altera_avalon_performance_counter.h>**
Parameters:	p—performance counter core base address.
Returns:	—
Description:	Macro `PERF_RESET()` stops and disables all counters, resetting them to 0.

36.6.2. PERF_START_MEASURING()

Prototype:	`PERF_START_MEASURING(p)`
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	**<altera_avalon_performance_counter.h>**
Parameters:	p—performance counter core base address.
Returns:	—
Description:	Macro `PERF_START_MEASURING()` starts the global counter, enabling the performance counter core. The behavior of individual section counters is controlled by `PERF_BEGIN()` and `PERF_END()`. `PERF_START_MEASURING()` defines the start of a global event, and increments the global event counter. This macro is a single write to the performance counter core.

36.6.3. PERF_STOP_MEASURING()

Prototype:	`PERF_STOP_MEASURING(p)`
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	**<altera_avalon_performance_counter.h>**

continued...

Parameters:	p—performance counter core base address.
Returns:	—
Description:	Macro `PERF_STOP_MEASURING()` stops the global counter, disabling the performance counter core. This macro is a single write to the performance counter core.

36.6.4. PERF_BEGIN()

Prototype:	`PERF_BEGIN(p,n)`
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	**<altera_avalon_performance_counter.h>**
Parameters:	p—performance counter core base address. n—counter section number. Section counter numbers start at 1. Do not refer to counter 0 in this macro.
Returns:	—
Description:	Macro `PERF_BEGIN()` starts the timer for a code section, defining the beginning of a section event, and incrementing the section event counter. If you subsequently use `PERF_STOP_MEASURING()` and `PERF_START_MEASURING()` to disable and re-enable the core, the section counter will resume. This macro is a single write to the performance counter core.

36.6.5. PERF_END()

Prototype:	`PERF_END(p,n)`
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	**<altera_avalon_performance_counter.h>**
Parameters:	p—performance counter core base address. n—counter section number. Section counter numbers start at 1. Do not refer to counter 0 in this macro.
Returns:	—
Description:	Macro `PERF_END()` stops timing a code section. The section counter does not run, regardless whether the core is enabled or not. This macro is a single write to the performance counter core.

36.6.6. perf_print_formatted_report()

Prototype:	`int perf_print_formatted_report (`
	`  void* perf_base,`
	`  alt_u32 clock_freq_hertz,`
	`  int num_sections,`
	`  char* section_name_1, ...`
	`  char* section_name_n)`

Thread-safe:	No.
Available from ISR:	No.
Include:	**<altera_avalon_performance_counter.h>**
Parameters:	`perf_base`—Performance counter core base address.

continued...

	`clock_freq_hertz`—Clock frequency. `num_sections`—The number of section counters to display. This must not exceed <instance_name>_HOW_MANY_SECTIONS. `section_name_1 ... section_name_n`—The section names to display. The number of section names varies depending on the number of sections to display.
Returns:	0
Description:	Function `perf_print_formatted_report()` reads the profiling results from the performance counter core, and prints a formatted summary table. This function disables all counters. However, for predictable results in a multi-threaded or interrupt environment, invoke `PERF_STOP_MEASURING()` when you reach the end of the code to be measured, rather than relying on `perf_print_formatted_report()`.

36.6.7. perf_get_total_time()

Prototype:	`alt_u64 perf_get_total_time(void* hw_base_address)`
Thread-safe:	No.
Available from ISR:	Yes.
Include:	**<altera_avalon_performance_counter.h>**
Parameters:	`hw_base_address`—base address of performance counter core.
Returns:	Aggregate global time in clock cycles.
Description:	Function `perf_get_total_time()` reads the raw global time. This is the aggregate time, in clock cycles, that the performance counter core has been enabled. This function has the side effect of stopping the counters.

36.6.8. perf_get_section_time()

Prototype:	`alt_u64 perf_get_section_time` `(void* hw_base_address, int which_section)`
Thread-safe:	No.
Available from ISR:	Yes.
Include:	**<altera_avalon_performance_counter.h>**
Parameters:	`hw_base_address`—performance counter core base address. `which_section`—counter section number.
Returns:	Aggregate section time in clock cycles.
Description:	Function `perf_get_section_time()` reads the raw time for a given section. This is the time, in clock cycles, that the section has been running. This function has the side effect of stopping the counters.

36.6.9. perf_get_num_starts()

Prototype:	`alt_u32 perf_get_num_starts` `(void* hw_base_address, int which_section)`
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	**<altera_avalon_performance_counter.h>**

continued...

Parameters:	`hw_base_address`—performance counter core base address. `which_section`—counter section number.
Returns:	Number of counter events.
Description:	Function `perf_get_num_starts()` retrieves the number of counter events (or times a counter has been started). If `which_section` = 0, it retrieves the number of global events (times the performance counter core has been enabled). This function does not stop the counters.

36.6.10. alt_get_cpu_freq()

Prototype:	`alt_u32 alt_get_cpu_freq()`
Thread-safe:	Yes.
Available from ISR:	Yes.
Include:	**<altera_avalon_performance_counter.h>**
Parameters:	
Returns:	CPU frequency in Hz.
Description:	Function `alt_get_cpu_freq()` returns the CPU frequency in Hz.

36.7. Performance Counter Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
June 2015	2015.06.12	Updated "Performance Counter Core Register Map" table.
July 2014	2014.07.24	Removed mention of SOPC Builder, updated to Platform Designer
December 2010	v10.1.0	Removed the “Device Support”, “Instantiating the Core in SOPC Builder”, and “Referenced Documents” sections.
July 2010	v10.0.0	Updated `perf_print_formatted_report()` to remove the restriction on using small C library.
November 2009	v9.1.0	No change from previous release.
March 2009	v9.0.0	No change from previous release.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	v8.0.0	Updated the parameter description of the function `perf_print_formatted_report()`.

37. Vectored Interrupt Controller Core

37.1. Core Overview

The ability to process interrupt events quickly and to handle large numbers of interrupts can be critical to many embedded systems. The Vectored Interrupt Controller (VIC) is designed to address these requirements. The VIC can provide interrupt performance four to five times better than the Nios II processor's default internal interrupt controller (IIC). The VIC also allows expansion to a virtually unlimited number of interrupts, through daisy chaining.

The vectored interrupt controller (VIC) core serves the following main purposes:

- Provides an interface to the interrupts in your system
- Reduces interrupt overhead
- Manages large numbers of interrupts

  The VIC offers high-performance, low-latency interrupt handling. The VIC prioritizes interrupts in hardware and outputs information about the highest-priority pending interrupt. When external interrupts occur in a system containing a VIC, the VIC determines the highest priority interrupt, determines the source that is requesting service, computes the requested handler address (RHA), and provides information, including the RHA, to the processor.

The VIC core contains the following interfaces:

- Up to 32 interrupt input ports per VIC core
- One Avalon Memory-Mapped (Avalon-MM) agent interface to access the internal control status registers (CSR)
- One Avalon Streaming (Avalon-ST) interface output interface to pass information about the selected interrupt
- One optional Avalon-ST interface input interface to receive the Avalon-ST output in systems with daisy-chained VICs

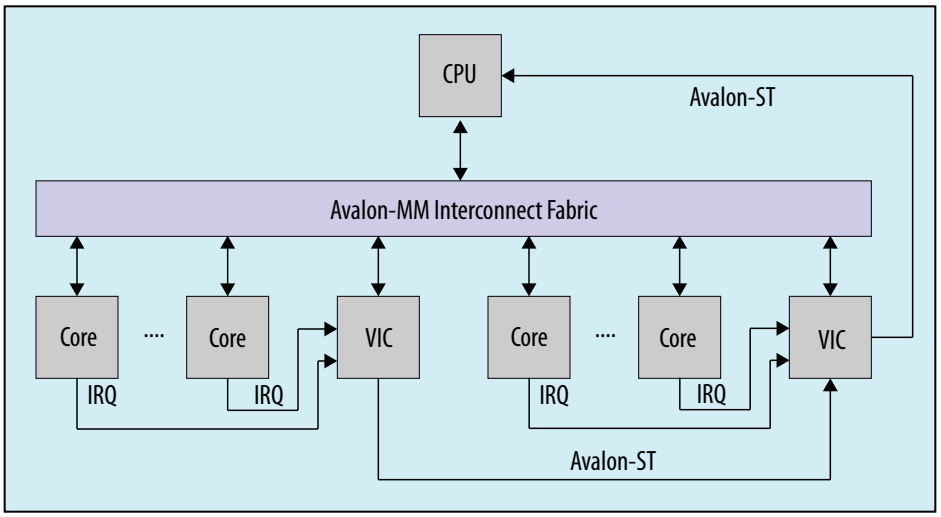
  The **Sample System Layout** Figure below outlines the basic layout of a system containing two VIC components.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Figure 111. Sample System Layout

The VIC core provides the following features:



To use the VIC, the processor in your system needs to have a matching Avalon-ST interface to accept the interrupt information, such as the Nios II processor's external interrupt controller interface.

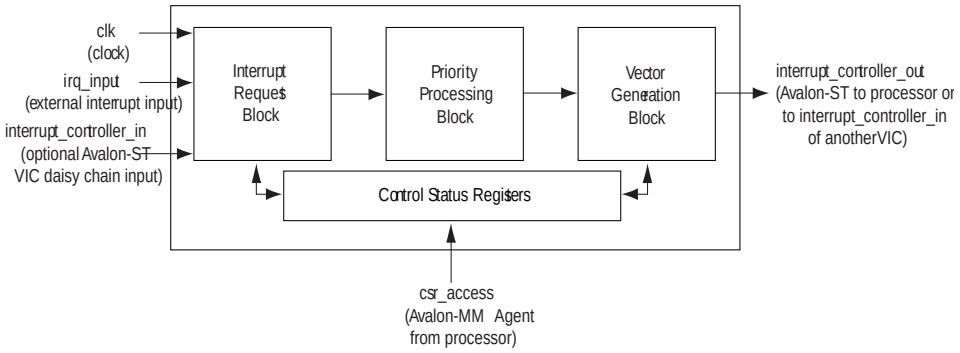
The characteristics of each interrupt port are configured via the Avalon-MM agent interface. When you need more than 32 interrupt ports, you can daisy chain multiple VICs together.

- Separate programmable requested interrupt level (RIL) for each interrupt
- Separate programmable requested register set (RRS) for each interrupt, to tell the interrupt handler which processor register set to use
- Separate programmable requested non-maskable interrupt (RNMI) flag for each interrupt, to control whether each interrupt is maskable or non-maskable
- Software-controlled priority arbitration scheme

  The VIC core is Platform Designer ready and integrates easily into any Platform Designer generated system. For the Nios II processor, Intel provides Hardware Abstraction Layer (HAL) driver routines for the VIC core. Refer to **to Intel FPGA HAL Software Programming Model** section for HAL support details.

37.2. Functional Description

Figure 112. VIC Block Diagram



37.2.1. External Interfaces

The following sections describe the external interfaces for the VIC core.

37.2.1.1. clk

`clk` is a system clock interface. This interface connects to your system's main clock source. The interface's signals are `clk` and `reset_n`.

37.2.1.2. irq_input

`irq_input` comprises up to 32 single-bit, level-sensitive Avalon interrupt receiver interfaces. These interfaces connect to interrupt sources. There is one `irq` signal for each interface.

37.2.1.3. interrupt_controller_out

`interrupt_controller_out` is an Avalon-ST output interface, as defined in the **VIC Avalon-ST Interface Fields**, configured with a ready latency of 0 cycles. This interface connects to your processor or to the `interrupt_controller_in` interface of another VIC. The interface's signals are `valid` and `data`.

Table 370. interrupt_controller_out and interrupt_controller_in Parameters

Parameter	Value
Symbol width	45 bits
Ready latency	0 cycles

37.2.1.4. interrupt_controller_in

`interrupt_controller_in` is an optional Avalon-ST input interface, as defined in **VIC Avalon-ST Interface Fields**, configured with a ready latency of 0 cycles. Include this interface in the second, third, etc, VIC components of a daisy-chained

multiple VIC system. This interface connects to the `interrupt_controller_out` interface of the immediately-preceding VIC in the chain. The interface's signals are `valid` and `data`.

The `interrupt_controller_out` and `interrupt_controller_in` interfaces have identical Avalon-ST formats so you can daisy chain VICs together in Platform Designer when you need more than 32 interrupts. `interrupt_controller_out` always provides valid data and cannot be back-pressured.

Table 371. VIC Avalon-ST Interface Fields

44				...					...			13	12-7	6	5-0
RHA(37)													RRS (38)	RNMI (38)	RIL(38)

37.2.1.5. csr_access

`csr_access` is a VIC CSR interface consisting of an Avalon-MM agent interface. This interface connects to the data host of your processor. The interface's signals are `read`, `write`, `address`, `readdata`, and `writedata`.

Table 372. csr_access Parameters

Parameter	Value
Read wait	1 cycle
Write wait	0 cycles
Ready latency	1 cycles

For information about the Avalon-MM agent and Avalon-ST interfaces, refer to the *Avalon Interface Specifications*.

Related Information

Avalon Interface Specifications

37.2.2. Functional Blocks

The following main design blocks comprise the VIC core:

- Interrupt request block
- Priority processing block
- Vector generation block

37.2.2.1. Interrupt Request Block

The interrupt request block controls the input interrupts, providing functionality such as setting interrupt levels, setting the per-interrupt programmable registers, masking interrupts, and managing software-controlled interrupts. You configure the number of interrupt input ports when you create the component. Refer to **Parameters** section for configuration options.

(37) RHA contains the 32-bit address of the interrupt handling routine.

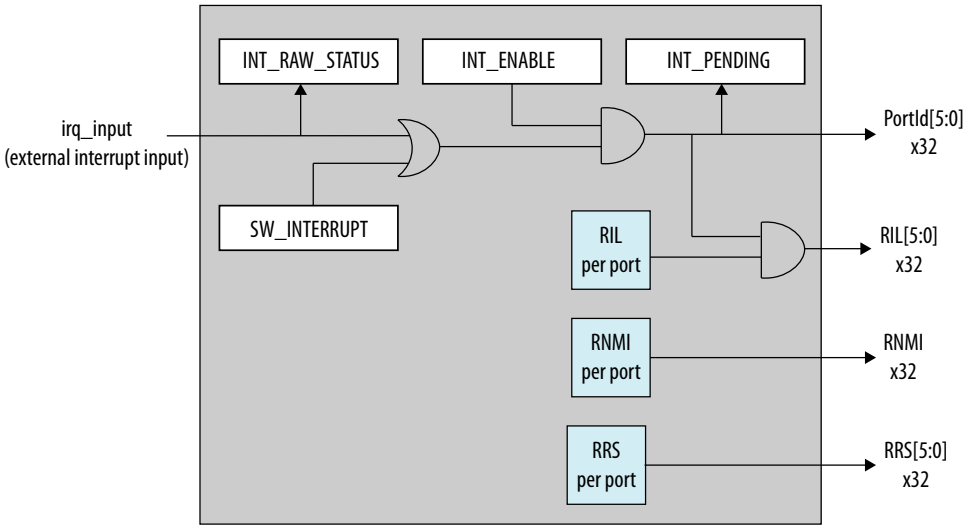
(38) Refer to The **INT_CONFIG Register Map** Table for a description of this field.

This block contains the majority of the VIC CSRs. The CSRs are accessed via the Avalon-MM agent interface.

Optional output from another VIC core can also come into the interrupt request block. Refer to the **Daisy Chaining VIC Cores** section for more information.

Each interrupt can be driven either by its associated `irq_input` signal (connected to a component with an interrupt source) or by a software trigger controlled by a CSR (even when there is no interrupt source connected to the `irq_input` signal).

Figure 113. Interrupt Request Block



37.2.2.2. Priority Processing Block

The priority processing block chooses the interrupt with the highest priority. The block receives information for each interrupt from the interrupt request block and passes information for the highest priority interrupt to the vector generation block.

The interrupt request with the numerically-largest requested interrupt level field (RIL) has priority. If multiple interrupts are pending with the same numerically-largest RIL, the numerically-lowest IRQ index of those interrupts has priority.

The RIL is a programmable interrupt level per port. An RIL value of zero disables the interrupt. You configure the bit width of the RIL when you create the component. Refer to the **Parameters** section for configuration options.

For more information about the RIL, refer to the `INT_CONFIG` register in the *Register Map* section of this chapter.

Related Information

Register Maps on page 455

37.2.2.3. Vector Generation Block

The vector generation block receives information for the highest priority interrupt from the priority processing block. The vector generation block uses the port identifier passed from the priority processing block along with the vector base address and bytes per vector programmed in the CSRs during software initialization to compute the RHA.

Table 373. RHA Calculation

RHA = (port identifier x bytes per vector) + vector base address

The information then passes out of the vector generation block and the VIC using the Avalon-ST interface. Refer to the **VIC Avalon-ST Interface Fields** table for details about the outgoing information. The output from the VIC typically connects to a processor or another VIC, depending on the design.

37.2.3. Daisy Chaining VIC Cores

You can create a system with more than 32 interrupts by daisy chaining multiple VIC cores together. This is done by connecting the `interrupt_controller_out` interface of one VIC to the optional `interrupt_controller_in` interface of another VIC. For information about enabling the optional input interface, refer to the **Parameters** section.

For performance reasons, always directly connect VIC components. Do not include other components between VICs.

When daisy chain input comes into the VIC, the priority processing block considers the daisy chain input along with the hardware and software interrupt inputs from the interrupt request block to determine the highest priority interrupt. If the daisy chain input has the highest RIL value, then the vector generation block passes the daisy chain port values unchanged directly out of the VIC.

You can daisy chain VICs with fewer than 32 interrupt ports. The number of daisy chain connections is only limited to the hardware and software resources. Refer to the **Latency Information** section for details about the impact of multiple VICs.

Intel recommends setting the RIL width to the same value in all daisy-chained VIC components. If your RIL widths are different, wider RILs from upstream VICs are truncated.

37.2.4. Latency Information

The latency of an interrupt request traveling through the VIC is the sum of the delay through each of the blocks. Clock delays in the interrupt request block and the vector generation block are constants. The clock delay in the priority processing block varies depending on the total number of interrupt ports.

Table 374. Default Interrupt Latencies

Number of Interrupt Ports	Interrupt Request Block Delay	Priority Processing Block Delay	Vector Generation Block Delay	Total Interrupt Latency
1	1 cycle	0 cycles	1 cycle	2 cycles
2 – 4	1 cycle	1 cycle	1 cycle	3 cycles
5 – 16	1 cycle	2 cycles	1 cycle	4 cycles
17 – 32	1 cycle	3 cycles	1 cycle	5 cycles

When daisy-chaining multiple VICs, interrupt latency increases as you move through the daisy chain away from the processor. For best performance, assign interrupts with the lowest latency requirements to the VIC connected directly to the processor.

37.3. Register Maps

The VIC core CSRs are accessible through the Avalon-MM interface. Software can configure the core and determine current status by accessing the registers.

Each register has a 32-bit interface that is not byte-enabled. You must access these registers with a host that is at least 32 bits wide.

Table 375. Control Status Registers

Offset	Register Name	Access	Reset Value	Description
0 – 31	INT_CONFIG<n>	R/W	0	There are 32 interrupt configuration registers (INT_CONFIG0 – INT_CONFIG31). Each register contains fields to configure the behavior of its corresponding interrupt. If an interrupt input does not exist, reading the corresponding register always returns zero, and writing is ignored. Refer to the **INT_CONFIG Register Map** table for the INT_CONFIG register map.
32	INT_ENABLE	R/W	0	The interrupt enable register. INT_ENABLE holds the enabled status of each interrupt input. The 32 bits of the register map to the 32 interrupts available in the VIC core. For example, bit 5 corresponds to IRQ5. (39) Interrupts that are not enabled are never considered by the priority processing block, even when the interrupt input is asserted. This applies to both maskable and non-maskable interrupts.
33	INT_ENABLE_SET	W	0	The interrupt enable set register. Writing a 1 to a bit in INT_ENABLE_SET sets the corresponding bit in INT_ENABLE. Writing a 0 to a bit has no effect. Reading from this register always returns 0. (39)
34	INT_ENABLE_CLR	W	0	The interrupt enable clear register. Writing a 1 to a bit in INT_ENABLE_CLR clears corresponding bit in INT_ENABLE. Writing a 0 to a bit has no effect. Reading from this register always returns 0. (39)
35	INT_PENDING	R	0	The interrupt pending register. INT_PENDING shows the pending interrupts. Each bit corresponds to one interrupt input. If an interrupt does not exist, reading its corresponding INT_PENDING bit always returns 0, and writing is ignored.

continued...

Offset	Register Name	Access	Reset Value	Description
				Bits in INT_PENDING are set in the following ways: An external interrupt is asserted at the VIC interface and the corresponding INT_ENABLE bit is set. An SW_INTERRUPT bit is set and the corresponding INT_ENABLE bit is set. INT_PENDING bits remain set as long as either condition applies. Refer to the **Interrupt Request Block** for details. (39)
36	INT_RAW_STATUS	R	0	The interrupt raw status register. INT_RAW_STATUS shows the unmasked state of the interrupt inputs. If an interrupt does not exist, reading the corresponding INT_RAW_STATUS bit always returns 0, and writing is ignored. A set bit indicates an interrupt is asserted at the interface of the VIC. The interrupt is asserted to the processor only when the corresponding bit in the interrupt enable register is set. (39)
37	SW_INTERRUPT	R/W	0	The software interrupt register. SW_INTERRUPT drives the software interrupts. Each interrupt is ORed with its external hardware interrupt and then enabled with INT_ENABLE. Refer to the **Interrupt Request Block** for details. (39)
38	SW_INTERRUPT_SET	W	0	The software interrupt set register. Writing a 1 to a bit in SW_INTERRUPT_SET sets the corresponding bit in SW_INTERRUPT. Writing a 0 to a bit has no effect. Reading from this register always returns 0. (39)
39	SW_INTERRUPT_CLR	W	0	The software interrupt clear register. Writing a 1 to a bit in SW_INTERRUPT_CLR clears the corresponding bit in SW_INTERRUPT. Writing a 0 to a bit has no effect. Reading from this register always returns 0.
40	VIC_CONFIG	R/W	0	The VIC configuration register. VIC_CONFIG allows software to configure settings that apply to the entire VIC. Refer to the **VIC_CONFIG Register Map** table for the VIC_CONFIG register map.

continued...

(39) This register contains a 1-bit field for each of the 32 interrupt inputs. When the VIC is configured for less than 32 interrupts, the corresponding 1-bit field for each unused interrupts is tied to zero. Reading these locations always returns 0, and writing is ignored. To determine which interrupts are present, write the value 0xffffffff to the register and then read the register contents. Any bits that return zero do not have an interrupt present.

Offset	Register Name	Access	Reset Value	Description
41	VIC_STATUS	R	0	The VIC status register. VIC_STATUS shows the current status of the VIC. Refer to the **VIC_STATUS Register Map** table for the VIC_STATUS register map.
42	VEC_TBL_BASE	R/W	0	The vector table base register. VEC_TBL_BASE holds the base address of the vector table in the processor's memory space. Because the table must be aligned on a 4-byte boundary, bits 1:0 must always be 0.
43	VEC_TBL_ADDR	R	0	The vector table address register. VEC_TBL_ADDR provides the RHA for the IRQ value with the highest priority pending interrupt. If no interrupt is active, the value in this register is 0. If daisy chain input is enabled and is the highest priority interrupt, the vector table address register contains the RHA value from the daisy chain input interface.

Table 376. The INT_CONFIG Register Map

Bits	Field Name	Access	Reset Value	Description
0:5	RIL	R/W	0	The requested interrupt level field. RIL contains the interrupt level of the interrupt requesting service. The processor can use the value in this field to determine if the interrupt is of higher priority than what the processor is currently doing.
6	RNMI	R/W	0	The requested non-maskable interrupt field. RNMI contains the non-maskable interrupt mode of the interrupt requesting service. When 0, the interrupt is maskable. When 1, the interrupt is non-maskable.
7:12	RRS	R/W	0	The requested register set field. RRS contains the number of the processor register set that the processor should use for processing the interrupt. Software must ensure that only register values supported by the processor are used.
13:31	Reserved			

For expanded definitions of the terms in the **INT_CONFIG Register Map** table, refer to the Exception Handling chapter of the *Nios II Software Developer's Handbook*.

Table 377. The VIC_CONFIG Register Map

Bits	Field Name	Access	Reset Value	Description
0:2	VEC_SIZE	R/W	0	The vector size field. VEC_SIZE specifies the number of bytes in each vector table entry. VEC_SIZE is encoded as log2 (number of words) - 2. Namely: 0—4 bytes per vector table entry 1—8 bytes per vector table entry 2—16 bytes per vector table entry 3—32 bytes per vector table entry 4—64 bytes per vector table entry 5—128 bytes per vector table entry 6—256 bytes per vector table entry 7—512 bytes per vector table entry
3	DC	R/W	0	The daisy chain field. DC serves the following purposes: Enables and disables the daisy chain input interface, if present. Write a 1 to enable the daisy chain interface; write a 0 to disable it.

continued...

Bits	Field Name	Access	Reset Value	Description
				Detects the presence of the daisy chain input interface. To detect, write a 1 to DC and then read DC. A return value of 1 means the daisy chain interface is present; 0 means the daisy chain interface is not present.
4:31	Reserved			

Table 378. The VIC_STATUS Register Map

Bits	Field Name	Access	Reset Value	Description
0:5	HI_PRI_IRQ	R	0	The highest priority interrupt field. HI_PRI_IRQ contains the IRQ number of the active interrupt with the highest RIL. When there is no active interrupt (IP is 0), reading from this field returns 0. When the daisy chain input is enabled and it is the highest priority interrupt, then the value read from this field is 32. Bit 5 always reads back 0 when the daisy chain input is not present.
6:30	Reserved			
31	IP	R	0	The interrupt pending field. IP indicates when there is an interrupt ready to be serviced. A 1 indicates an interrupt is pending; a 0 indicates no interrupt is pending.

Related Information

- Exception Handling
- Priority Processing Block on page 453

37.4. Parameters

Generation-time parameters control the features present in the hardware.The table below lists and describes the parameters you can configure.

Table 379. Parameters for VIC Core

Parameter	Legal Values	Default	Description
Number of interrupts	1 – 32	8	Specifies the number of `irq_input` interrupt interfaces.
RIL width	1 – 6	4	Specifies the bit width of the requested interrupt level.
Daisy chain enable	True / False	False	Specifies whether or not to include an input interface for daisy chaining VICs together.
Override Default Interrupt Signal Latency	True/False	False	Allows manual specification of the interrupt signal latency.
Manual Interrupt Signal Latency	2 – 5	2	Specifies the number of cycles it takes to process incoming interrupt signals.

Because multiple VICs can exist in a single system, Platform Designer assigns a unique interrupt controller identification number to each VIC generated.

Keep the following considerations in mind when connecting the core in your Platform Designer system:

- The CSR access interface (`csr_access`) connects to a data host port on your processor.
- The daisy chain input interface (`interrupt_controller_in`) is only visible when the daisy chain enable option is on.
- The interrupt controller output interface (`interrupt_controller_out`) connects either to the EIC port of your processor, or to another VIC's daisy chain input interface (`interrupt_controller_in`).
- For Platform Designer interoperability, the VIC core includes an Avalon-MM host port. This host interface is not used to access memory or peripherals. Its purpose is to allow peripheral interrupts to connect to the VIC in Platform Designer. The port must be connected to an Avalon-MM agent to create a valid Platform Designer system. Then at system generation time, the unused host port is removed during optimization. The most simple solution is to connect the host port directly into the CSR access interface (`csr_access`).
- Platform Designer automatically connects interrupt sources when instantiating components. When using the provided HAL device driver for the VIC, daisy chaining multiple VICs in a system requires that each interrupt source is connected to exactly one VIC. You need to manually remove any extra connections.

37.5. Intel FPGA HAL Software Programming Model

The Intel-provided driver implements a HAL device driver that integrates with a HAL board support package (BSP) for Nios II systems. HAL users should access the VIC core via the familiar HAL API.

37.5.1. Software Files

The VIC driver includes the following software files. These files provide low-level access to the hardware and drivers that integrate with the Nios II HAL BSP. Application developers should not modify these files.

- `altera_vic_regs.h`—Defines the core's register map, providing symbolic constants to access the low-level hardware.
- `altera_vic_funnel.h`, `altera_vic_irq.h`, `altera_vic_macros.h`—Define the prototypes and macros necessary for the VIC driver.
- `altera_vic.c`, `altera_vic_irq_init.c`, `altera_vic_isr_register.c`, `altera_vic_sw_intr.c`, `altera_vic_set_level.c`, **altera_vic_funnel_non_preemptive_nmi.S**, **altera_vic_funnel_non_preemptive.S**, and **altera_vic_funnel_preemptive.S**—Provide the code that implements the VIC driver.
- **altera_**<name>**_vector_tbl.S**—Provides a vector table file for each VIC in the system. The BSP generator creates these files.

37.5.2. Macros

Macros to access all of the registers are defined in **altera_vic_regs.h.** For example, this file includes macros to access the `INT_CONFIG` register, including the following macros:

```
#define IOADDR_ALTERA_VIC_INT_CONFIG(base, irq)
   __IO_CALC_ADDRESS_NATIVE(base, irq)
#define IORD_ALTERA_VIC_INT_CONFIG(base, irq) IORD(base, irq)
#define IOWR_ALTERA_VIC_INT_CONFIG(base, irq, data) IOWR(base, irq,
data)
#define ALTERA_VIC_INT_CONFIG_RIL_MSK (0x3f)
#define ALTERA_VIC_INT_CONFIG_RIL_OFST (0)
#define ALTERA_VIC_INT_CONFIG_RNMI_MSK (0x40)
#define ALTERA_VIC_INT_CONFIG_RNMI_OFST (6)
#define ALTERA_VIC_INT_CONFIG_RRS_MSK (0x1f80)
#define ALTERA_VIC_INT_CONFIG_RRS_OFST (7)
```

For a complete list of predefined macros and utilities to access the VIC hardware, refer to the following files:

- <install_dir>**\ip\altera\altera_vectored_interrupt_controller\top\inc\altera_vic_regs.h**
- <install_dir>**\ip\altera\altera_vectored_interrupt_controller\top\HAL\inc\altera_vic_funnel.h**
- <install_dir>**\ip\altera\altera_vectored_interrupt_controller\top\HAL\inc\altera_vic_irq.h**

37.5.3. Data Structure

Example 10. Device Data Structure

```
#define ALT_VIC_MAX_INTR_PORTS          (32)

typedef struct alt_vic_dev
{
   void       *base;                   /* Base address of VIC */
   alt_u32    intr_controller_id;      /* Interrupt controller ID */
   alt_u32    num_of_intr_ports;       /* Number of interrupt ports */
   alt_u32    ril_width;               /* RIL width */
   alt_u32    daisy_chain_present;     /* Daisy-chain input present */
   alt_u32    vec_size;                /* Vector size */
   void       *vec_addr;               /* Vector table base address */
   alt_u32    int_config[ALT_VIC_MAX_INTR_PORTS]; /* INT_CONFIG settings
                                                     for each interrupt */

} alt_vic_dev;
```

37.5.4. VIC API

The VIC device driver provides all the routines required of an Intel FPGA HAL external interrupt controller (EIC) device driver. The following functions are required by the Nios II enhanced HAL interrupt API:

- alt_ic_isr_register()
- alt_ic_irq_enable()
- alt_ic_irq_disable()
- alt_ic_irq_enabled()

  These functions write to the register map to change the setting or read from the register map to check the status of the VIC component through a memory-mapped address.

  For detailed descriptions of these functions, refer to the to the HAL API Reference chapter of the *Nios II Software Developer's Handbook*.

  The table below lists the API functions specific to the VIC core and briefly describes each. Details of each function follow the table.

Table 380. Function List

Name	Description
`alt_vic_sw_interrupt_set()`	Sets the corresponding bit in the SW_INTERRUPT register to enable a given interrupt via software.
`alt_vic_sw_interrupt_clear()`	Clears the corresponding bit in the SW_INTERRUPT register to disable a given interrupt via software.
`alt_vic_sw_interrupt_status()`	Reads the status of the SW_INTERRUPT register for a given interrupt.
`alt_vic_irq_set_level()`	Sets the interrupt level for a given interrupt.

Related Information

HAL API Reference

37.5.4.1. alt_vic_sw_interrupt_set()

Prototype:	int alt_vic_sw_interrupt_set(alt_u32 ic_id, alt_u32 irq)
Thread-safe:	No
Available from ISR:	No
Include:	**altera_vic_irq.h**, **altera_vic_regs**.h
Parameters:	ic_id—the interrupt controller identification number as defined in `system.h` irq—the interrupt value as defined in `system.h`
Returns:	Returns zero if successful; otherwise non-zero for one or more of the following reasons: The value in ic_id is invalid The value in irq is invalid
Description:	Triggers a single software interrupt

37.5.4.2. alt_vic_sw_interrupt_clear()

Prototype:	int alt_vic_sw_interrupt_clear(alt_u32 ic_id, alt_u32 irq)
Thread-safe:	No
Available from ISR:	Yes; if interrupt preemption is enabled, disable global interrupts before calling this routine.
Include:	**altera_vic_irq.h**, **altera_vic_regs**.h
Parameters:	ic_id—the interrupt controller identification number as defined in `system.h` irq—the interrupt value as defined in `system.h`
Returns:	Returns zero if successful; otherwise non-zero for one or more of the following reasons: The value in ic_id is invalid The value in irq is invalid
Description:	Clears a single software interrupt

37.5.4.3. alt_vic_sw_interrupt_status()

Prototype:	alt_u32 alt_vic_sw_interrupt_status(alt_u32 ic_id, alt_u32 irq)
Thread-safe:	No
Available from ISR:	Yes; if interrupt preemption is enabled, disable global interrupts before calling this routine.
Include:	**altera_vic_irq.h**, **altera_vic_regs**.h
Parameters:	ic_id—the interrupt controller identification number as defined in `system.h` irq—the interrupt value as defined in `system.h`
Returns:	Returns non-zero if the corresponding software trigger interrupt is active; otherwise zero for one or more of the following reasons: The corresponding software trigger interrupt is disabled The value in ic_id is invalid The value in irq is invalid
Description:	Checks the software interrupt status for a single interrupt

37.5.4.4. alt_vic_irq_set_level()

Prototype:	int alt_vic_irq_set_level(alt_u32 ic_id, alt_u32 irq, alt_u32 level)
Thread-safe:	No
Available from ISR:	No
Include:	**altera_vic_irq.h**, **altera_vic_regs**.h
Parameters:	ic_id—the interrupt controller identification number as defined in `system.h` irq—the interrupt value as defined in `system.h` level—the interrupt level to set
Returns:	Returns zero if successful; otherwise non-zero for one or more of the following reasons: The value in ic_id is invalid The value in irq is invalid The value in level is invalid
Description:	Sets the interrupt level for a single interrupt. Intel recommends setting the interrupt level only to zero to disable the interrupt or to the original value specified in your BSP. Writing any other value could violate the overlapping register set, priority level, and other design rules. Refer to the **VIC BSP Design Rules for to Intel FPGA HAL Implementation** section for more information.

37.5.5. Run-time Initialization

During system initialization, software configures the each VIC instance's control registers using settings specified in the BSP. The RIL, RRS, and RNMI fields are written into the interrupt configuration register of each interrupt port in each VIC. All interrupts are disabled until other software registers a handler using the alt_ic_isr_register() API.

37.5.6. Board Support Package

The BSP you generate for your Nios II system provides access to the hardware in your system, including the VIC. The VIC driver includes scripts that the BSP generator calls to get default interrupt settings and to validate settings during BSP generation. The Nios II BSP Editor provides a mechanism to edit these settings and generate a BSP for your Platform Designer design.

The generator produces a vector table file for each VIC in the system, named **altera_**<name>**_vector_tbl.S**. The vector table's source path is added to the BSP Makefile for compilation along with other VIC driver source code. Its contents are based on the BSP settings for each VIC's interrupt ports.

The VIC does not support runtime stack checking feature (**hal.enable_runtime_stack_checking**) in the BSP setting.

VIC BSP Settings

The VIC driver scripts provide settings to the BSP. The number and naming of these settings depends on your hardware system's configuration, specifically, the number of optional shadow register sets in the Nios II processor, the number of VIC controllers in the system, and the number of interrupt ports each VIC has.

Certain settings apply to all VIC instances in the system, while others apply to a specific VIC instance. Settings that apply to each interrupt port apply only to the specified interrupt port number on that VIC instance.

The remainder of this section lists details and descriptions of each VIC BSP setting.

37.5.6.1. altera_vic_driver.enable_preemption

Identifier:	ALTERA_VIC_DRIVER_ISR_PREEMPTION_ENABLED
Type:	BooleanDefineOnly
Default value:	1 when all components connected to the VICs support preemption. 0 when any of the connected components don't support preemption.
Destination file:	**system**.h
Description:	Enables global interrupt preemption (nesting). When enabled (set to 1), the macro `ALTERA_VIC_DRIVER_ISR_PREEMPTION_ENABLED` is defined in `system.h`. Two types of ISR preemption are available. This setting must be enabled along with other settings to enable specific types of preemption. All preemption settings are dependent on whether the device drivers in your BSP support interrupt preemption. For more information about preemption, refer to the **Exception Handling** chapter of the **Nios II Software Developer's Handbook**.
Occurs:	Once per VIC

37.5.6.2. altera_vic_driver.enable_preemption_into_new_register_set

Identifier:	ALTERA_VIC_DRIVER_PREEMPTION_INTO_NEW_REGISTER_SET_ENABLED
Type:	BooleanDefineOnly
Default value:	0
Destination file:	**system**.h
Description:	Enables interrupt preemption (nesting) if a higher priority interrupt is asserted while a lower priority ISR is executing, and that higher priority interrupt uses a different register set than the interrupt currently being serviced. When this setting is enabled (set to 1), the macro `ALTERA_VIC_DRIVER_ISR_PREEMPTION_INTO_NEW_REGISTER_SET_ENABLED` is defined in `system.h` and the Nios II `config.ANI` (automatic nested interrupts) bit is asserted during system software initialization. Use this setting to limit interrupt preemption to higher priority (RIL) interrupts that use a different register set than a lower priority interrupt that might be executing. This setting allows you to support some preemption while maintaining the lowest possible interrupt response time. However, this setting does not allow an interrupt at a higher priority (RIL) to preempt a lower priority interrupt if the higher priority interrupt is assigned to the same register set as the lower priority interrupt.
Occurs:	Once per VIC

37.5.6.3. altera_vic_driver.enable_preemption_rs_<n>

Identifier:	ALTERA_VIC_DRIVER_ENABLE_PREEMPTION_RS_<n>
Type:	Boolean
Default value:	0
Destination file:	**system**.h
Description:	Enables interrupt preemption (nesting) if a higher priority interrupt is asserted while a lower priority ISR is executing, for all interrupts that target the specified register set number. When this setting is enabled (set to 1), the vector table for each VIC utilizes a special interrupt funnel that manages preemption. All interrupts on all VIC instances assigned to that register set then use this funnel. When a higher priority interrupt preempts a lower priority interrupt running in the same register set, the interrupt funnel detects this condition and saves the processor registers to the stack before calling the higher priority ISR. The funnel code restores registers and allows the lower priority ISR to continue running once the higher priority ISR completes. Because this funnel contains additional overhead, enabling this setting increases interrupt response time substantially for all interrupts that target a register set where this type of preemption is enabled. Use this setting if you must guarantee that a higher priority interrupt preempts a lower priority interrupt, and you assigned multiple interrupts at different priorities to the same Nios II shadow register set.
Occurs:	Per register set; <n> refers to the register set number.

37.5.6.4. altera_vic_driver.linker_section

Identifier:	ALTERA_VIC_DRIVER_LINKER_SECTION
Type:	UnquotedString
Default value:	.text
Destination file:	**system**.h
Description:	Specifies the linker section that each VIC's generated vector table and each interrupt funnel link to. The memory device that the specified linker section is mapped to must be connected to both the Nios II instruction and data hosts in your Platform Designer system. Use this setting to link performance-critical code into faster memory. For example, if your system's code is in DRAM and you have an on-chip or tightly-coupled memory interface for interrupt handling code, assigning the VIC driver linker section to a section in that memory improves interrupt response time. For more information about linker sections and the Nios II BSP Editor, refer to the **Getting Started with the Graphical User Interface** chapter of the *Nios II Software Developer's Handbook*.
Occurs:	Once per VIC

37.5.6.5. altera_vic_driver.<name>.vec_size

Identifier:	<name>_VEC_SIZE
Type:	DecimalNumber
Default value:	16
Destination file:	**system**.h
Description:	Specifies the number of bytes in each vector table entry. Legal values are 16, 32, 64, 128, 256, and 512. The generated VIC vector tables in the BSP require a minimum of 16 bytes per entry. If you intend to write your own vector table or locate your ISR at the vector address, you can use a larger size. The vector table's total size is equal to the number of interrupt ports on the VIC instance multiplied by the vector table entry size specified in this setting.
Occurs:	Per instance; <name> refers to the component name you assign in Platform Designer.

37.5.6.6. altera_vic_driver.<name>.irq<n>_rrs

Identifier:	ALTERA_VIC_DRIVER_<name>_IRQ<n>_RRS
Type:	DecimalNumber
Default value:	Refer to the **Default Settings for RRS and RIL** section.
Destination file:	**system**.h
Description:	Specifies the RRS for the interrupt connected to the corresponding port. Legal values are 1 to the number of shadow register sets defined for the processor.
Occurs:	Per IRQ per instance; <name> refers to the VIC's name and <n> refers to the IRQ number that you assign in Platform Designer. Refer to Platform Designer to determine which IRQ numbers correspond to which components in your design.

37.5.6.7. altera_vic_driver.<name>.irq<n>_ril

Identifier:	ALTERA_VIC_DRIVER_<name>_IRQ<n>_RIL
Type:	DecimalNumber
Default value:	Refer to **Default Settings for RRS and RIL** section.
Destination file:	**system**.h
Description:	Specifies the RIL for the interrupt connected to the corresponding port. Legal values are 0 to 2RIL width -1.
Occurs:	Per IRQ per instance; <name> refers to the VIC's name and <n> refers to the IRQ number that you assign in Platform Designer. Refer to Platform Designer to determine which IRQ numbers correspond to which components in your design.

37.5.6.8. altera_vic_driver.<name>.irq<n>_rnmi

Identifier:	ALTERA_VIC_DRIVER_<name>_IRQ<n>_RNMI
Type:	Boolean
Default value:	0
Destination file:	**system**.h
Description:	Specifies whether the interrupt port is a maskable or non-maskable interrupt (NMI). Legal values are 0 and 1. When set to 0, the port is maskable. NMIs cannot be disabled in hardware and there are several restrictions imposed for the RIL and RRS settings associated with any interrupt with NNI enabled.
Occurs:	Per IRQ per instance; <name> refers to the VIC's name and <n> refers to the IRQ number that you assign in Platform Designer. Refer to Platform Designer to determine which IRQ numbers correspond to which components in your design.

37.5.6.9. Default Settings for RRS and RIL

The default assignment of RRS and RIL values for each interrupt assumes interrupt port 0 on the VIC instance attached to your processor is the highest priority interrupt, with successively lower priorities as the interrupt port number increases. Interrupt ports on other VIC instances connected through the first VIC's daisy chain interface are assigned successively lower priorities.

To make effective use of the VIC interrupt setting defaults, assign your highest priority interrupts to low interrupt port numbers on the VIC closest to the processor. Assign lower priority interrupts and interrupts that do not need exclusive access to a shadow register set, to higher interrupt port numbers, or to another daisy-chained VIC.

The following steps describe the algorithm for default RIL assignment:

1. The formula $2^{\text{RIL width}}$ -1 is used to calculate the maximum RIL value.
2. interrupt port 0 on the VIC connected to the processor is assigned the highest possible RIL.
3. The RIL value is decremented and assigned to each subsequent interrupt port in succession until the RIL value is 1.
4. The RILs for all remaining interrupt ports on all remaining VICs in the chain are assigned 1.

   The following steps describe the algorithm for default RRS assignment:

5. The highest register set number is assigned to the interrupt with the highest priority.
6. Each subsequent interrupt is assigned using the same method as the default RIL assignment.

   For example, consider a system with two VICs, VIC0 and VIC1. Each VIC has an RIL width of 3, and each has 4 interrupt ports. VIC0 is connected to the processor and VIC1 to the daisy chain interface on VIC0. The processor has 3 shadow register sets.

Table 381. Default RRS and RIL Assignment Example

VIC	IRQ	RRS	RIL
0	0	3	7
0	1	2	6
0	2	1	5
0	3	1	4
1	0	1	3
1	1	1	2
1	2	1	1
1	3	1	1

37.5.6.10. VIC BSP Design Rules for Intel FPGA HAL Implementation

The VIC BSP settings allow for a large number of combinations. This list describes some basic design rules to follow to ensure a functional BSP:

- Each component's interrupt interface in your system should only be connected to one VIC instance per processor.
- The number of shadow register sets for the processor must be greater than zero.
- RRS values must always be greater than zero and less than or equal to the number of shadow register sets.
- RIL values must always be greater than zero and less than or equal to the maximum RIL.
- All RILs assigned to a register set must be sequential to avoid a higher priority interrupt overwriting contents of a register set being used by a lower priority interrupt.

Note: The Nios II BSP Editor uses the term "overlap condition" to refer to non-sequential RIL assignments.

- NMIs cannot share register sets with maskable interrupts.
- NMIs must have RILs set to a number equal to or greater than the highest RIL of any maskable interrupt. When equal, the NMIs must have a lower logical interrupt port number than any maskable interrupt.
- The vector table and funnel code section's memory device must connect to a data host and an instruction host.
- NMIs must use funnels with preemption disabled.
- When global preemption is disabled, enabling preemption into a new register set or per-register-set preemption might produce unpredictable results. Be sure that all interrupt service routines (ISR) used by the register set support preemption.
- Enabling register set preemption for register sets with peripherals that don't support preemption might result in unpredictable behavior.

37.5.6.11. RTOS Considerations

BSPs configured to use a real time operating system (RTOS) might have additional software linked into the HAL interrupt funnel code using the `ALT_OS_INT_ENTER` and `ALT_OS_INT_EXIT` macros. The exact nature and overhead of this code is RTOS-specific. Additional code adds to interrupt response and recovery time. Refer to your RTOS documentation to determine if such code is necessary.

37.6. Implementing the VIC in Platform Designer

This section describes how to incorporate one or more VICs in your Platform Designer system, and how to support the VIC in software.

37.6.1. Adding VIC Hardware

When you add a VIC to your Platform Designer system, you must perform the following high-level tasks:

1. Add the EIC interface to your Nios II processor core
2. Optionally add shadow register sets to your Nios II processor core (required if you intend to use HAL interrupt support)
3. Add and parameterize one or more VIC components
4. Connect interrupt sources to the VIC component(s)

37.6.1.1. Adding the EIC Interface Shadow Register Set

This section describes how to add the EIC interface and shadow register sets to a Nios II processor core in Platform Designer, through the parameter editor interface.

1. In Platform Designer, double-click the Nios II processor to open the parameter editor interface.
2. Enable the EIC interface on the Nios II processor by selecting it in the **Interrupt Controller** list in the **Advanced Features** tab, as shown in the figure below.

   There are two options for **Interrupt Controller: Internal** and **External**. If you select **Internal**, the processor is implemented with the internal interrupt controller. Select **External** to implement the processor with an EIC interface.

   Note: When you implement the EIC interface, you must connect an EIC, such as the VIC. Failure to connect an EIC results in a Platform Designer error.
3. Select the desired number of shadow register sets. In the **Number of shadow register sets** list, select the number of register sets that matches your system performance goals.
4. Click **Finish** to exit from the Nios II parameter editor interface . Notice that the processor shows an unconnected `interrupt_controller_in` Avalon-ST sink, as shown in the figure below.

Figure 114. Configuring the Interrupt Controller and Shadow Register Sets

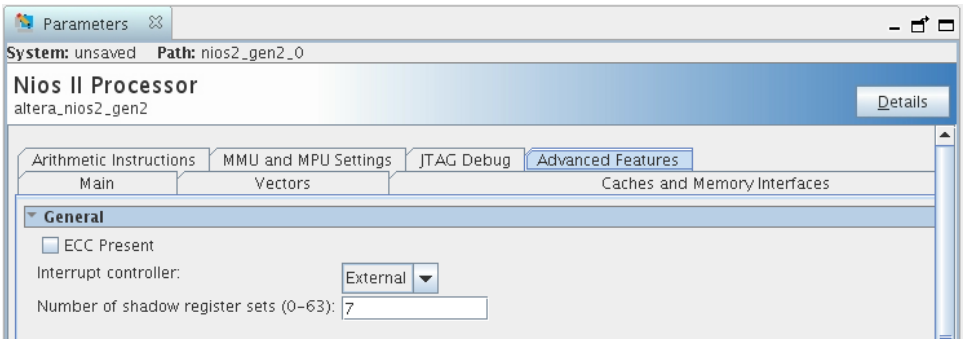


Figure 115. Nios II Processor with EIC Interface



Shadow register sets reduce the context switching overhead associated with saving and restoring registers, which can otherwise be significant. If possible, add one shadow register set for each interrupt that requires high performance.

37.6.1.2. VIC Instantiation, Parameterization, and Connection

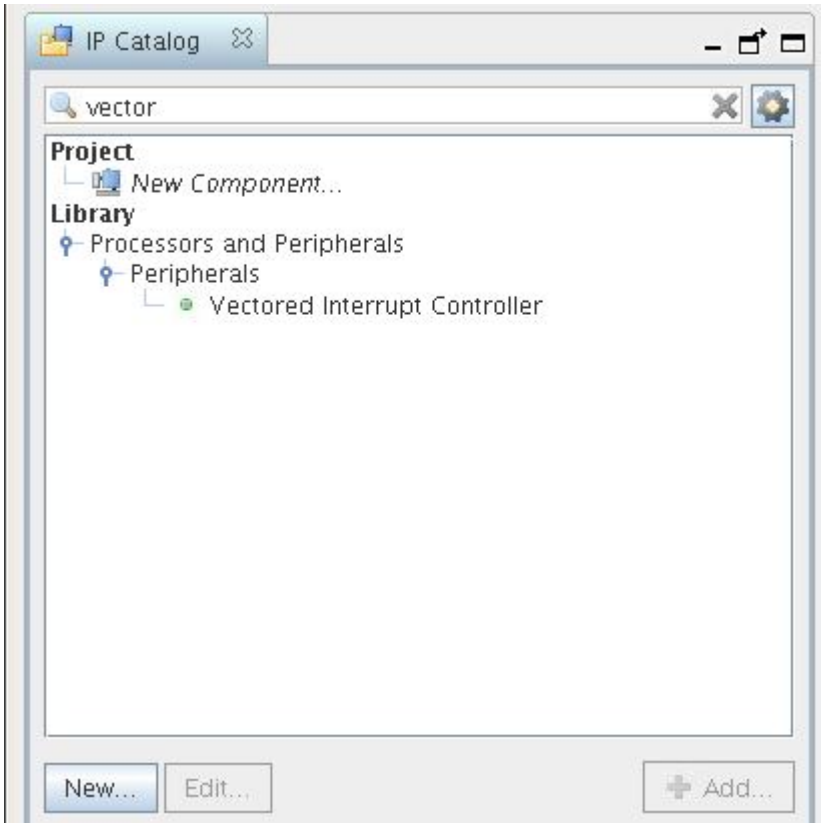
After you add the EIC interface and shadow register set(s) to the Nios II processor, you must instantiate and parameterize the VIC in your Platform Designer system.

37.6.1.2.1. Instantiation

To instantiate a VIC in your Platform Designer system, execute the following steps:

1. Browse to the **IP Catalog** window in Platform Designer.
2. Type "vector" in the search box. The interface hides all components except the VIC, as shown in the figure below.
3. Double click the Vectored Interrupt Controller component to add this component to your Platform Designer System.

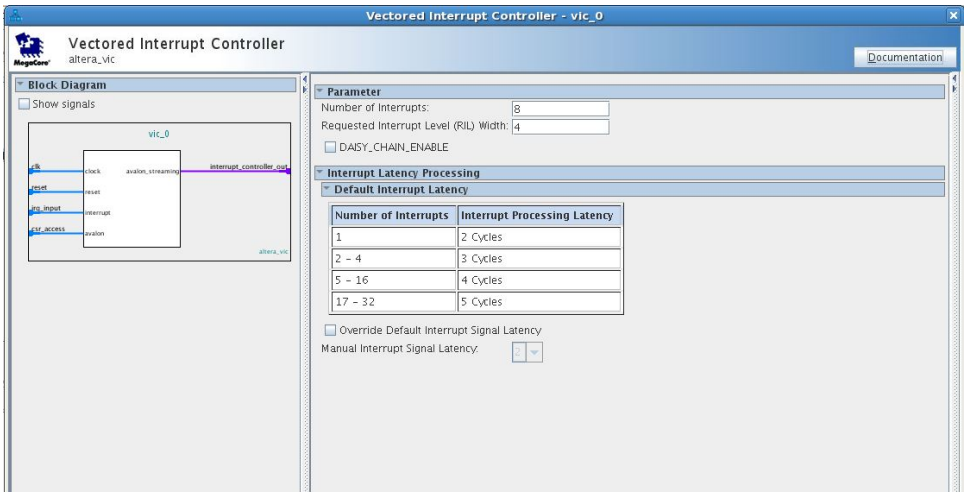
Figure 116. Vectored Interrupt Controller Component



37.6.1.2.2. Parameterization

When you add the VIC to your system, the Vectored Interrupt Controller interface appears as shown below.

Figure 117. Vectored Interrupt Controller Parameterization



The VIC interface allows you to specify the following options:

- **Number of Interrupts**—The number of interrupts your VIC must support.
- **Requested Interrupt Level (RIL) Width**—The number of bits allocated to represent the interrupt level for each interrupt.
- **DAISY_CHAIN_ENABLE**—Allows the VIC to daisy chain to another EIC. Turn on this option if you want to support multiple VICs in your system.

  Note: Study the VIC Daisy-Chain example that accompanies this document for a usage example.
- **Override Default Interrupt Signal Latency**—Allows manual specification of the interrupt signal latency.
- **Manual Interrupt Signal Latency**—Specifies the number of cycles it takes to process the incoming interrupt signals.

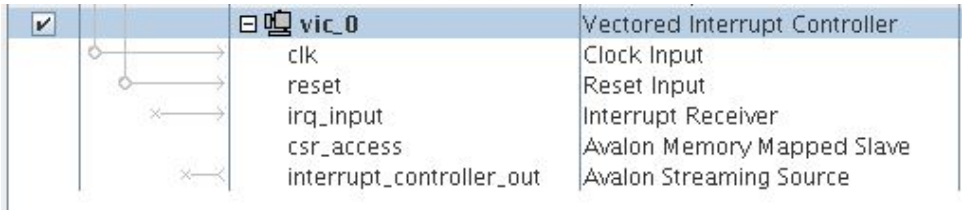
When you have finished parameterizing the VIC, click **Finish** to instantiate the component in your Platform Designer system.

37.6.1.2.3. VIC Connections

When you have added the VIC to your system, it appears in Platform Designer as shown below.

Note: If you have enabled daisy chaining, Platform Designer adds an Avalon-ST sink, called `interrupt_controller_in`, to the VIC.

Figure 118. VIC Interfaces

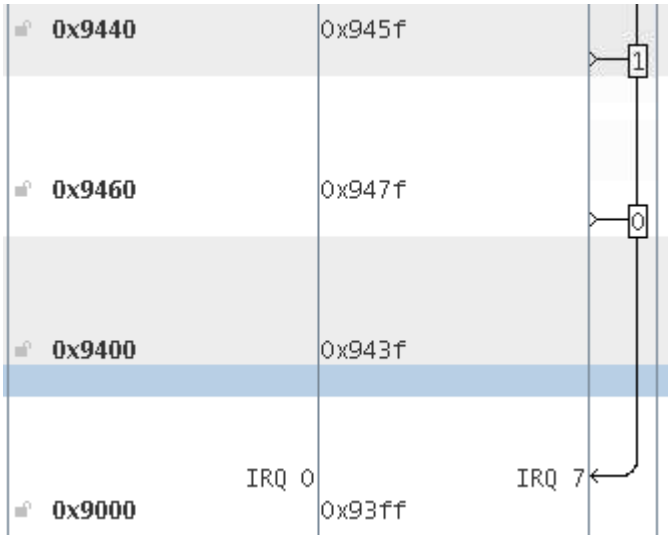


After adding a VIC to the Platform Designer system, you must parameterize the VIC and the EIC interface at the system level. Immediately after you add the VIC, several error messages appear. Resolve these error messages by executing the following actions in any order:

- Connect the VIC's `interrupt_controller_out` Avalon-ST source to the `interrupt_controller_in` Avalon-ST sink on either the Nios II processor or the next VIC in a daisy-chained configuration.
- Connect the Nios II processor's `data_master` Avalon-MM ports to the `csr_access` Avalon-MM agent port.
- Assign an interrupt number for each interrupt-based component in the system, as shown below. This step connects each component to an interrupt port on the VIC.

Note: If your system contains more than one EIC connected to a single processor, you must ensure that each component is connected to an interrupt port on only one EIC.

Figure 119. Assigning Interrupt Numbers



When you use the HAL VIC driver, the driver makes a default assignment from register sets to interrupts. The default assignment makes some assumptions about interrupt priorities, based on how devices are connected to the VIC.

Note: To make effective use of the VIC interrupt setting defaults, assign your highest priority interrupts to low interrupt port numbers on the VIC closest to the processor.

37.6.2. Software for VIC

If you write an interrupt handler for a system based on the VIC component, you must use the HAL enhanced interrupt API to register the handler and control its runtime environment. The enhanced interrupt API provides a number of functions for use with EICs, including the VIC. This section describes a subset of the functions in the enhanced interrupt API.

For information about the enhanced interrupt API, refer to "Interrupt Service Routines" in the Exception Handling chapter of the *Nios II Software Developer's Handbook*.

In particular, this section shows how to code a driver so that it supports both the enhanced API and the legacy API. This must include testing for the presence of the enhanced API, and conditionally calling the appropriate function.

Related Information

Interrupt Service Routines

37.6.2.1. alt_ic_isr_register() versus alt_irq_register()

The enhanced API function `alt_ic_isr_register()` is very similar to the legacy function `alt_irq_register()`, with a few important differences. The differences between these two functions are best understood by examining the code in Registering an ISR with Both APIs on page 474. This example registers a timer interrupt in either the legacy API or the enhanced API, whichever is implemented in the board support package (BSP). The example is taken directly from the example code accompanying this document.

Example 11. Registering an ISR with Both APIs

```
#ifdef ALT_ENHANCED_INTERRUPT_API_PRESENT
void timer_interrupt_latency_init (void* base, alt_u32 irq_controller_id,
alt_u32 irq)
{
  /* Register the interrupt */
  alt_ic_isr_register(irq_controller_id, irq, timer_interrupt_latency_irq,
base, NULL);
  /* Start timer */
  IOWR_ALTERA_AVALON_TIMER_CONTROL(base, ALTERA_AVALON_TIMER_CONTROL_ITO_MSK
  | ALTERA_AVALON_TIMER_CONTROL_START_MSK);
}
#else
void timer_interrupt_latency_init (void* base, alt_u32 irq)
{
  /* Register the interrupt */
  alt_irq_register(irq, base, timer_interrupt_latency_irq);
  /* Start timer */
  IOWR_ALTERA_AVALON_TIMER_CONTROL(base, ALTERA_AVALON_TIMER_CONTROL_ITO_MSK
  | ALTERA_AVALON_TIMER_CONTROL_START_MSK);
}
#endif
```

The first line of Registering an ISR with Both APIs on page 474 detects whether the BSP implements the enhanced interrupt API. If the enhanced API is implemented, the `timer_interrupt_latency_init()` function calls the enhanced function. If not, `timer_interrupt_latency_init()` reverts to the legacy interrupt API function.

For an explanation of how the Nios II Software Build Tools select which API to implement in a BSP, refer to "Interrupt Service Routines" in the Exception Handling chapter of the *Nios II Software Developer's Handbook*.

Enhanced Function alt_ic_isr_register() on page 475 shows the function prototype for `alt_ic_isr_register()`, which registers an ISR in the enhanced API. The interrupt controller identifier (for argument `ic_id`) and the interrupt port number (for argument `irq`) are defined in **system.h**.

Example 12. Enhanced Function alt_ic_isr_register()

```
extern int alt_ic_isr_register(alt_u32 ic_id,
  alt_u32 irq,
  alt_isr_func isr,
  void *isr_context,
  void *flags);
```

For comparison, Legacy Function alt_irq_register() on page 475 shows the function prototype for `alt_irq_register()`, which registers an ISR in the legacy API.

Example 13. Legacy Function alt_irq_register()

```
extern int alt_irq_register (alt_u32 id,
  void* context,
  alt_isr_func handler);
```

The arguments passed into `alt_ic_isr_register()` are slightly different from those passed into `alt_irq_register()`. The table below compares the arguments to the two functions.

Table 382. Arguments to alt_ic_isr_register() versus alt_irq_register()

alt_ic_isr_register() Argument	Purpose	alt_irq_register() Argument
`alt_u32 ic_id`	Unique interrupt controller ID as defined in **system.h**.	—
`alt_u32 irq`	Interrupt request (IRQ) number as defined in **system.h**.	`alt_u32 id`
`alt_isr_func isr`	Interrupt service routine (ISR) function pointer	`handler`
`void* isr_context`	Optional pointer to a component-specific data structure.	`context`
`void* flags`	Reserved. Other EIC implementations might use this argument.	None

There are other significant differences between the legacy interrupt API and the enhanced interrupt API. Some of these differences impact the ISR body itself. Notably, the two APIs employ completely different interrupt preemption models. The example code accompanying this document illustrates many of the differences.

For further information about the other functions in the HAL interrupt APIs, refer to the Exception Handling and HAL API Reference chapters of the *Nios II Software Developer's Handbook*.

Related Information

- Exception Handling
- HAL API Reference

37.7. Example Designs

This section provides a brief description of the example designs provided with this document to demonstrate the usage of the VIC. Additionally, this section provides instructions for running the software examples on the Cyclone V SoC development kit.

37.7.1. Example Description

The example designs are provided in a file called **VIC_collateral_cv.zip**.

Table 383. Example Designs in VIC_collateral_cv.zip

Example Name	Folder Name	Description
VIC Basic	VIC_Example	A single VIC
VIC Daisy-Chain	VIC_DaisyChain_Example	Two daisy-chained VICs
VIC Table-Resident	VIC_ISRnVectorTable_Example	VIC with ISR located in vector table
IIC	VIC_noVIC_Example	IIC example, for comparison with the VIC examples

The top-level folder in **VIC_collateral_cv.zip**, called **VIC_collateral_cv**, contains the following files:

- **run_sw.sh**—Shell script to run one, several or all of the examples
- **README.txt**—Describes the **.zip** file contents

Figure 120. VIC Basic Example



Figure 121. VIC Daisy-Chain Example



The IIC design is the same as the VIC Basic design, with the VIC and the EIC interface replaced by the IIC. The VIC Table-Resident design is identical to the VIC Basic design.

In each example, the software uses timers in conjunction with performance counters to measure the interrupt performance. Each example's software calculates the performance and sends the results to stdout.

VIC_collateral_cv.zip includes a script, **run_sw.sh**, to run one, several, or all of the example. **run_sw.sh** downloads the SRAM Object File (**.sof**) and the Executable and Linkable Format File (**.elf**) for each example, and executes the code on the Cyclone V SoC, for the examples that you specify on the command line.

Note: **run_sw.sh** assumes that you have only one JTAG download cable connected to your host computer. If you have multiple JTAG cables, you must modify **run_sw.sh** to specify the cable connected to your Cyclone V SoC development kit.

Related Information

- Nios II Processor Support
- VIC_collateral_cv.zip

37.7.2. Example Usage

Initially, Intel recommends that you run each example design as distributed, to see the example's performance on your own hardware. Thereafter, you can modify any of the examples to investigate the VIC's performance options, or customize the code for you application.

Execute the following steps to run each example design:

1. Power up your Cyclone V SoC board.
2. Connect the USB cable.
3. Unzip the **VIC_collateral_cv.zip** file to a working directory, expanding folder names.

   Note: The path name to your working directory must not contain any spaces.
4. In a Nios II Command Shell, change to the top-level directory, **VIC_collateral_cv**.
5. At the command prompt, type the following command:

   ```
   ./run_sw.sh
   ```

   The script shows a list of options.
6. Run **run_sw.sh** again, using a command-line option that specifies the example you would like to run, or to run all of the examples. VIC Example on page 479 shows a sample session.

   The **run_sw.sh** script performs the following steps:

   a. Parses the command line argument(s) to determine which example(s) to run
   b. Downloads the **.sof** for the selected example
   c. Downloads the **.elf** for the selected example
   d. Starts **nios2-terminal** to capture the software's output

37.7.3. Software Description

The software for the various example designs is very similar. For example, the difference between the software for the VIC Basic example and the software for the IIC example is the `printf()` call that generates the output to the terminal.

All of the software performs the following steps:

1. Configures the timer used for measurement purposes
2. Registers an interrupt service routine (ISR)
3. Sets a global variable to `0xfeedface`

4. Starts the performance counter to measure the interrupt time
5. Waits for the ISR to set the global variable to `0xfacefeed`
6. Stops the performance counter and computes the interrupt time

The VIC Daisy-Chain example performs the measurement for both VICs connected in the daisy chain, shown in Figure 121 on page 477.

In all these design examples, the GCC compiler in Nios II SBT tool is set to optimization level 2. Also, some settings are modified during BSP generation in order to reduce the code size. All these setting can be found in the create-this-bsp script included in the design example. Note that the number of clock cycles shows in these design examples will be differ from this document if the setting is different.

For details about how the VIC Table-Resident example code works, refer to "Positioning the ISR in the Vector Table". For details about performance counter usage in the example software, refer to "Latency Measurement with the Performance Counter".

Example 14. VIC Example

```
$ ./run_sw.sh --VIC_Example

Running software...



Running for VIC_Example
/cygdrive/c/altera/VIC_Example/software_examples/app /cygdrive/c/altera
Searching for SOF file:
in ../../
  VIC_Example.sof

Info: ************************************************************************
Info: Running Quartus II 64-Bit Programmer
Info: Command: quartus_pgm --no_banner --mode=jtag -o p;C:/altera/VIC_Example/VI
C_Example.sof@2
Info (213045): Using programming cable "USB-BlasterII [USB-1]"
Info (213011): Using programming file C:/altera/VIC_Example/VIC_Example.sof with
 checksum 0x0194989D for device 5CSXFC6D6F31@2
Info (209060): Started Programmer operation at Wed Mar 02 08:52:20 2016
Info (209016): Configuring device index 2
Info (209017): Device 2 contains JTAG ID code 0x02D020DD
Info (209007): Configuration succeeded -- 1 device(s) configured
Info (209011): Successfully performed operation(s)
Info (209061): Ended Programmer operation at Wed Mar 02 08:52:24 2016
Info: Quartus II 64-Bit Programmer was successful. 0 errors, 0 warnings
    Info: Peak virtual memory: 265 megabytes
    Info: Processing ended: Wed Mar 02 08:52:24 2016
    Info: Elapsed time: 00:00:06
    Info: Total CPU time (on all processors): 00:00:02
Using cable "USB-BlasterII [USB-1]", device 2, instance 0x00
Pausing target processor: OK
Initializing CPU cache (if present)
OK
Downloaded 5KB in 0.0s
Verified OK
Starting processor at address 0x00004020
nios2-terminal: connected to hardware target using JTAG UART on cable
nios2-terminal: "USB-BlasterII [USB-1]", device 2, instance 0
nios2-terminal: (Use the IDE stop button or Ctrl-C to terminate)

Starting VIC Example roundtrip performance test.

Interrupt Time:       53 clocks.

Sending EOT to force an exit.

nios2-terminal: exiting due to ^D on remote
/cygdrive/c/altera



Done...
```

Related Information

- Positioning the ISR in Vector Table on page 480
- Latency Measurement with the Performance Counter on page 481

37.7.4. Positioning the ISR in Vector Table

If have a critical ISR of small size, you can achieve the best performance by positioning the ISR code directly in the vector table. In this way, you eliminate the overhead of branching from the vector table through the HAL funnel to your ISR.This section describes how to modify the VIC Basic example software to create the VIC Table-Resident example. Use this example to ensure that you understand the steps. Then you can make the equivalent changes in your custom code.

Positioning an ISR in a vector table is an advanced and error-prone technique, not directly supported by the HAL. You must exercise great caution to ensure that the ISR code fits in the vector table entry. If your ISR overflows the vector table entry, it corrupts other entries in the vector table, and your entire interrupt handling system.

When locate your ISR in the vector table, it does not need to be registered. Do not call `alt_ic_isr_register()`, because it overwrites the contents of the vector table.

When the ISR is in the vector table, the HAL does not provide funnel code. Therefore, the ISR code must perform any context-switching actions normally handled by the funnel. Funnel context switching can include some or all of the following actions:

- Saving and restoring registers
- Managing preemption
- Managing the stack pointer

To create the fastest possible ISR, minimize or eliminate the context-switching actions your ISR must perform by conforming to the following guidelines:

- Write the ISR in assembly language
- Assign a shadow register set for the ISR's use
- Ensure that the ISR cannot be preempted by another ISR using the same register set. By default, preemption within a register set is disabled on the Nios II processor. You can also ensure this condition by giving the ISR exclusive access to its register set.

The VIC Table-Resident example requires modifying a BSP-generated file, **altera_vic1_vector_tbl.S**. If you regenerate the BSP after making these modifications, the Nios II Software Build Tools regenerate **altera_vic1_vector_tbl.S**, and your changes are overwritten.

Related Information

Software Description on page 478

37.7.4.1. Increase the Vector Table Entry Size

To insert the ISR in the vector table, you must increase the size of the vector entries so that your entire ISR fits in a vector table entry. Use the `altera_vic_driver.<vic_instance>.vec_size` BSP setting to adjust the vector

table entry size. On the Nios II Software Build Tools command line, you can manipulate this setting with the `--set` command-line option. You can also modify this setting in the Nios II BSP Editor.

In the VIC Table-Resident example, `<vic_instance>` is `VIC1` and `<size>` is set to 256 bytes.

37.7.4.2. Do Not Register the ISR

Remove the call to `alt_ic_isr_register()` for the interrupt that you place in the vector table. Replace it with an `alt_ic_irq_enable()` call. You must not call `alt_ic_isr_register()`, because it overwrites the contents of the vector table, destroying the body of your ISR.

37.7.4.3. Insert ISR in Vector Table

In the VIC Table-Resident example included with this document, the ISR code is in a file called **vector.h** in the BSP folder.

To insert this code in the vector table, execute the following steps:

1. Generate the BSP by running the **create-this-bsp** script.
2. Modify **altera_vic1_vector_tbl.S** as shown in the example below.

Example 15. Modifications to Intel FPGA_vic1_vector_tbl.S

```
#include "altera_vic_funnel.h"
#include "vector.h"                  /* ADD THIS LINE MANUALLY */
    .section .text
    .align 2
    .globl VIC1_VECTOR_TABLE
VIC1_VECTOR_TABLE:
    MY_ISR 256                     /* THIS LINE REPLACES THE FIRST VECTOR TABLE
ENTRY */
    ALT_SHADOW_NON_PREEMPTIVE_INTERRUPT 256
    ALT_SHADOW_NON_PREEMPTIVE_INTERRUPT 256
    ALT_SHADOW_NON_PREEMPTIVE_INTERRUPT 256
    ALT_SHADOW_NON_PREEMPTIVE_INTERRUPT 256
```

After completion of these steps, build the software, run it, and observe the reported interrupt time. This example is about 18 clock cycles faster than the unmodified VIC Basic example.

Some variation is likely for reasons discussed in “Real-Time Latency Concerns”.

Related Information

Real Time Latency Concerns on page 482

37.7.5. Latency Measurement with the Performance Counter

The Intel Quartus Prime enables you to make fast, accurate performance measurements. All examples included with this document use the Performance Counter component to measure interrupt latency.

The examples execute the following steps to measure the total time spent to service an interrupt:

1. Initialize a global variable, `interrupt_watch_value`, to a known value, `0xfeedface`.
2. Set up a timer interrupt, registering an ISR that sets `interrupt_watch_value` to `0xfacefeed`.
3. Start the timer.
4. Wait in a `while()` loop until `interrupt_watch_value` becomes `0xfacefeed`.
5. Immediately after exiting the `while()` loop, stop the performance counter, compute clock cycles and display the calculated value on `stdout`.

You can use similar methods to determine the real-time interrupt latencies in your system.

Related Information

- Software Description on page 478
- Real Time Latency Concerns on page 482

37.8. Advanced Topics

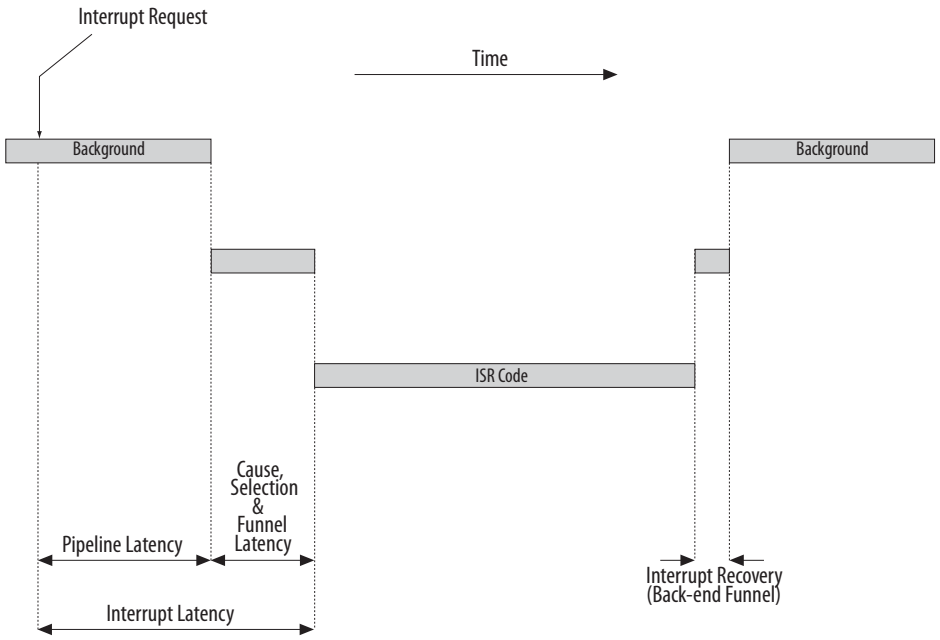
This section presents several topics that are useful for advanced interrupt handling.

37.8.1. Real Time Latency Concerns

This section presents an overview of interrupt latency, the elements that combine to determine interrupt latency, and methods for measuring it. The following elements comprise interrupt latency:

- Pipeline latency
- Cause latency
- Selection latency
- Funnel latency
- Compiler-related latency

Figure 122. The Elements of Interrupt Latency



This section summarizes each element of latency and describes how to measure latency. The accompanying example designs use the performance counter core to capture all of the timing measurements. Performance counter core usage is described in "Latency Measurement with the Performance Counter".

Related Information

- Insert ISR in Vector Table on page 481
- Latency Measurement with the Performance Counter on page 481

37.8.1.1. Pipeline Latency

Pipeline latency is defined as the number of clock cycles between an interrupt signal being asserted and the execution of the first instruction at the exception vector. It can vary widely, depending on the type of memory the processor is executing from and the impact of other host ports in your hardware. Theoretically, this time could be infinite if an ill-behaved host port blocks the processor from accessing memory, freezing the processor.

37.8.1.2. Cause Latency

Cause latency is the time required for the processor to identify an exception as a hardware interrupt. With an EIC, such as the VIC, the cause latency is zero because each hardware interrupt has a dedicated interrupt vector address, separate from the software exception vector address.

37.8.1.3. Selection Latency

Selection latency is the time required for the system to transfer control to the correct interrupt vector, depending on which interrupt is triggered. The selection latency with the VIC component depends on the number of interrupts that it services. The table below outlines selection latency on a single VIC as a function of the number of interrupts.

Table 384. The Components of VIC Latency

Total Number of Interrupts	Interrupt Request Clock Delay (clocks)	Priority Processing Clock Delay (clocks)	Vector Generation Clock Delay (clocks)	Total Interrupt Latency (clocks)
1	2	0	1	3
2—4	2	1	1	4
5—16	2	2	1	5
17—32	2	3	1	6

37.8.1.4. Funnel Latency

Funnel latency is the time required for the interrupt funnel to switch context. Funnel latency can include saving and restoring registers, managing preemption, and managing the stack pointer. Funnel latency depends on the following factors:

- Whether a separate interrupt stack is used
- The number of clock cycles required for load and store instructions
- Whether the interrupt requires switching to a different register set
- Whether the interrupt is preempting another interrupt within the same register set
- Whether preemption within the register set is allowed

Preemption within the register set requires special attention. The HAL VIC driver provides special funnel code if an interrupt is allowed to preempt another interrupt assigned to the same register set. In this case, the funnel incurs additional overhead to save and restore the register contents. When creating the BSP, you can control preemption within the register set by using the VIC driver's `altera_vic_driver_enable_preemption_rs_<n>` setting.

Note: With tightly-coupled memory, the Nios II processor can execute a load or store instruction in 1 clock cycle. With onchip memory, not tightly-coupled, the processor requires two clock cycles.

Table 385. Single Stack HAL latency

Funnel Type	Clock Cycles Required for Load or Store	
	1	2
Shadow register set, preemption within the register set disabled	10	13
Shadow register set, preemption within the register set enabled	42 Same register set (`sstatus.SRS=0`)	64 Same register set (`sstatus.SRS=0`)
	26 Different register set (`sstatus.SRS=1`)	32 Different register set (`sstatus.SRS=1`)

Table 386. Separate Interrupt Stack HAL Latency

Funnel Type	Clock Cycles Required for Load or Store	
	1	2
Shadow register set, preemption within the register set disabled	11 Not preempting another interrupt (`sstatus.IH=0`)	14 Not preempting another interrupt (`sstatus.IH=0`)
	12 Preempting another interrupt (`sstatus.IH=1`)	15 Preempting another interrupt (`sstatus.IH=1`)
Shadow register set, preemption within the register set enabled	42 Same register set (`sstatus.SRS=0`)	64 Same register set (`sstatus.SRS=0`)
	27 • Different register set (`sstatus.SRS=1`) • Not preempting another interrupt (`sstatus.IH=0`)	33 • Different register set (`sstatus.SRS=1`) • Not preempting another interrupt (`sstatus.IH=0`)
	28 • Different register set (`sstatus.SRS=1`) • Preempting another interrupt (`sstatus.IH=1`)	34 • Different register set (`sstatus.SRS=1`) • Preempting another interrupt (`sstatus.IH=1`)

In the tables above, notice that the lowest latencies occur under the following conditions:

- A different register set—Shadow register set switch; the ISR runs in a different register set from the interrupted task, eliminating any need to save or restore registers.
- Preemption (nesting) within the register set disabled.

Conversely, the highest latencies occur under the following conditions:

- The same register set—No shadow register set switch; the ISR runs in the same register set as the interrupted task, requiring the funnel code to save and restore registers.
- Preemption within the register set enabled.

Of these two important factors, preemption makes the largest difference in latencies. With preemption disabled, much lower latencies occur regardless of other factors.

37.8.1.5. Compiler-Related Latency

The GNU C compiler creates a prologue and epilogue for many C functions, including ISRs. The prologue and epilogue are code sequences that take care of housekeeping tasks, such as saving and restoring context for the C runtime environment. The time required for the prologue and epilogue is called compiler-related latency.

The C compiler generates a prologue and epilogue as needed. If compiler optimization is enabled, and the routine is compact, with few local variables, the prologue and epilogue are usually omitted. You can determine whether a prologue and epilogue are generated by examining the function's assembly code.

Compiler latency normally has only a minor impact on overall interrupt servicing performance. If you are concerned about compiler latency, you have two options:

- Enable compiler optimizations, and simplify your ISR, minimizing local variables.
- Write your ISR in assembly language.

37.8.2. Software Interrupt

Software can trigger any VIC interrupt by writing to the appropriate VIC control and status register (CSR). Software can trigger the interrupt connected to any hardware interrupt source, as well as interrupts that are not connected to hardware (software-only interrupts).

Triggering an interrupt from software is useful for debugging. Software can control exactly when an interrupt is triggered, and measure the system's interrupt response.

You can use a software-only interrupt to re-prioritize an interrupt. An ISR that responds to a high-priority hardware interrupt can perform the minimum processing required by the hardware, and then trigger a software-only interrupt at a lower priority level to complete the interrupt processing.

The following functions are available for managing software interrupts:

- `alt_vic_sw_interrupt_set()`
- `alt_vic_sw_interrupt_clear()`
- `alt_vic_sw_interrupt_status()`

The implementations of these functions are in **bsp/hal/drivers/src/altera_vic_sw_intr.c** after you generate the BSP.

Note: You must define a value for the interrupt number in `SOFT_IRQ`.

Example 16. Registering a Software Interrupt

```
alt_ic_isr_register(
  VIC1_INTERRUPT_CONTROLLER_ID,
  SOFT_IRQ,
  soft_interrupt_latency_irq,
  NULL, NULL)
```

Example 17. Registering a Timer Interrupt (for Comparison)

```
alt_ic_isr_register(
  LATENCY_TIMER_IRQ_INTERRUPT_CONTROLLER_ID,
  LATENCY_TIMER_IRQ,
  timer_interrupt_latency_irq,
  LATENCY_TIMER_BASE,
  NULL);
```

The following code generates a software interrupt:

```
alt_vic_sw_interrupt_set(VIC1_INTERRUPT_CONTROLLER_ID, SOFT_IRQ);
```

37.9. Vectored Interrupt Controller Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0	• Fixed related links and typos. • Implemented editorial enhancements.

Date	Version	Changes
May 2016	2016.05.03	Sections Added: • Implementing VIC in Platform Designer • Example Designs • Advanced Topics
November 2015	2015.11.06	Updated: • Table 372 on page 452 • Table 374 on page 455 • Table 379 on page 459
December 2013	v13.1.0	Updated the INT_ENABLE register description.
December 2010	v10.1.0	Added a note to to state that the VIC does not support the runtime stack checking feature in BSP setting. Removed the "Device Support", "Instantiating the Core in SOPC Builder", and "Referenced Documents" sections.
July 2010	v10.0.0	No change from previous release.
November 2009	v9.1.0	Initial release.

38. Avalon-ST Data Pattern Generator and Checker Cores

38.1. Core Overview

The data generation and monitoring solution for Avalon Streaming (Avalon-ST) interfaces consists of two components: a data pattern generator core that generates data patterns and sends it out on an Avalon-ST interface, and a data pattern checker core that receives the same data and checks it for correctness.

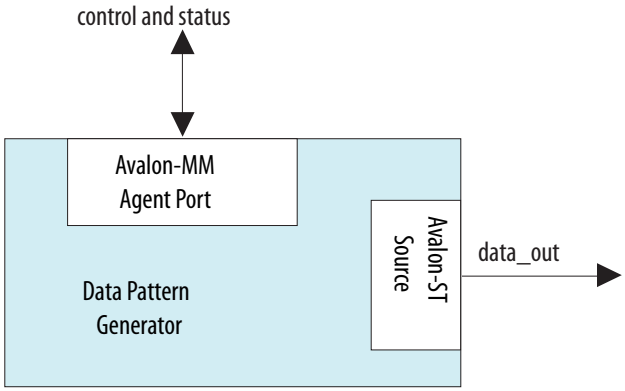
38.2. Data Pattern Generator

This section describes the hardware structure and functionality of the data pattern generator core.

38.2.1. Functional Description

The data pattern generator core accepts commands to generate and drive data onto a parallel Avalon-ST source interface.

Figure 123. Data Pattern Generator Core Block Diagram



You can configure the width of the output data signal to either 32-bit or 40-bit when instantiating the core.

You can configure this core to output 8-bit or 10-bit wide symbols. By default, the core generates 4 symbols per beat, which outputs 32-bit or 40-bit wide data to the Avalon-ST interfaces, respectively. The core's data format endianness is the most significant symbol first within a beat and the most significant bit first within a symbol. For example, when you configure the output data to 32-bit, bit 31 is the first data bit, followed by bit 30, and so forth. This interface's endianness may change in future versions of the core.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

For smaller data widths, you can use the Avalon-ST Data Format Adapter for data width adaptation. The Avalon-ST Data Format Adapter converts the output from 4 symbols per beat, to 2 or 1 symbol per beat. In this way, the 32-bit output of the core can be adapted to a 16-bit or 8-bit output and the 40-bit output can be adapted to a 20-bit or 10-bit output.

For more information about the Avalon-ST Data Format Adapter, refer to Platform Designer User Guide.

Control and Status Interface

The control and status interface is an Avalon-MM agent that allows you to enable or disable the data generation. This interface also provides the run-time ability to choose data pattern and inject an error into the data stream.

Output Interface

The output interface is a parallel Avalon-ST interface. You can configure the data width at the output interface to suit your requirements.

Supported Data Patterns

The following data patterns are supported in the following manner, per beat. When the core is disabled or in idle state, the default pattern generated on the data output is 0×5555 (for 32-bit data width) or 0×55555 (for 40-bit data width).

Table 387. Supported Data Patterns (Binary Encoding)

Pattern	32-bit	40-bit
PRBS-7	PRBS in parallel	PRBS in parallel
PRBS-15	PRBS in parallel	PRBS in parallel
PRBS-23	PRBS in parallel	PRBS in parallel
PRBS-31	PRBS in parallel	PRBS in parallel
High Frequency	10101010 x 4	1010101010 x 4
Low Frequency	11110000 x 4	1111100000 x 4
Note to **Table 29–1** : 1. All PRBS patterns are seeded with 11111111.		

This core does not support custom data patterns.

Inject Error

Errors can be injected into the data stream by controlling the `Inject Error` register bits in the register map (refer to the **Inject Error Field Descriptions** table). When the inject error bit is set, one bit of error is produced by inverting the LSB of the next data beat.

If the inject error bit is set before the core starts generating the data pattern, the error bit is inserted in the first output cycle.

The `Inject Error` register bit is automatically reset after the error is introduced in the pipeline, so that the next error can be injected.

Preamble Mode

The preamble mode is used for synchronization or word alignment. When the preamble mode is set, the preamble control register sends the preamble character a specified number of times before the selected pattern is generated, so the word alignment block in the receiver can determine the word boundary in the bit stream.

The number of bits (`Numbits`) determines the number of cycles to output the preamble character in the preamble mode. You can set the number of bits (`Numbits`) in the preamble control register. The default setting is 0 and the maximum value is 255 bits. This mode can only be set when the data pattern generation core is disabled.

38.2.2. Configuration

You can configure your core by setting the following parameters in the Platform Designer:

- You can configure the input interface of the data pattern generator core using the following parameter:

  ST_DATA_W — The width of the input data signal that the data pattern checker core supports. Valid values are 32, 40, 50, 64, 66, 80, and 128.
- You can add a bypass interface to register and output the input data through a bypass port using the following parameter:

  Enable Bypass Interface — Select this option to enable this interface. By default, this interface is disabled.
- The data pattern generator core supports two types of interface: Avalon-ST and Conduit interface. You can select either of them using the following parameter:

  Enable Avalon Interface — Select this option to enable Avalon interface. By default this interface is enabled. De-select this option to enable Conduit interface.
- You can enable frequency counter by selecting the following parameter:

  Enable Frequency Counter
- The following parameter determines the synchronization depth for clock crossing from Avalon-MM clock domain to Avalon-ST clock domain:

  CROSS_CLK_SYNC_DEPTH — Default value is 2. Valid values are => 2.

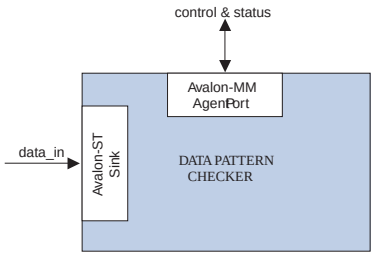
38.3. Data Pattern Checker

This section describes the hardware structure and functionality of the data pattern checker core.

38.3.1. Functional Description

The data pattern checker core accepts data via an Avalon-ST sink interface, checks it for correctness against the same predetermined pattern used by the data pattern generator core or other PRBS generators to produce the data, and reports any exceptions to the control interface.

Figure 124. Data Pattern Checker



You can configure the width of the output data signal to either 32-bit or 40-bit when instantiating the core. The chosen data width is not configurable during run time.

You can configure this core to output 8-bit or 10-bit wide symbols. By default, the core generates 4 symbols per beat, which outputs 32-bit or 40-bit wide data to the Avalon-ST interfaces, respectively. The core's data format endianness is the most significant symbol first within a beat and the most significant bit first within a symbol. For example, when you configure the output data to 32-bit, bit 31 is the first data bit, followed by bit 30, and so forth. This interface's endianness may change in future versions of the core.

If you configure the width of the output data to 32-bit, the core inputs four 8-bit wide symbols per beat. To achieve an 8-bit and 16-bit data width, you can use the Avalon-ST Data Format Adapter component to convert 4 symbols per beat to 1 or 2 symbols per beat.

Similarly, if you configure the width of the output data to 40-bit, the core inputs four 10-bit wide symbols per beat. The 10-bit and 20-bit input can be achieved by switching from 4 symbols per beat to 1 and 2 symbols per beat.

Control and Status Interface

The control and status interface is an Avalon-MM agent that allows you to enable or disable the pattern checking. This interface also provides the run-time ability to choose the data pattern and read the status signals.

Input Interface

The input interface is a parallel Avalon-ST interface. You can configure the data width at this interface to suit your requirements.

Supported Data Patterns

The following data patterns are supported in the following manner, per beat. When the core is disabled or in idle state, the default pattern generated on the data output is 0×5555 (for 32-bit data width) or 0×55555 (for 40-bit data width).

Table 388. Supported Data Patterns (Binary Encoding)

Pattern	32-bit	40-bit
PRBS-7	PRBS in parallel	PRBS in parallel
PRBS-15	PRBS in parallel	PRBS in parallel

continued...

Pattern	32-bit	40-bit
PRBS-23	PRBS in parallel	PRBS in parallel
PRBS-31	PRBS in parallel	PRBS in parallel
High Frequency	10101010 x 4	1010101010 x 4
Low Frequency	11110000 x 4	1111100000 x 4

Lock

The lock bit in the status register is asserted when 40 consecutive bits of correct data are received. The lock bit is deasserted and the receiver loses the lock when 40 consecutive bits of incorrect data are received.

Bit and Error Counters

The core has two 64-bit internal counters to keep track of the number of bits and number of error bits received. A snapshot has to be executed to update the `NumBits` and `NumErrors` registers with the current value from the internal counters.

A counter reset can be executed to reset both the registers and internal counters. If the counters are not being reset and the core is enabled, the internal counters continues the increment base on their current value.

The internal counters only start to increment after a lock has been acquired.

38.3.2. Configuration

You can configure your core by setting the following parameters in the Platform Designer:

- You can configure the input interface of the data pattern checker core using the following parameter:

  ST_DATA_W — The width of the input data signal that the data pattern checker core supports. Valid values are 32, 40, 50, 64, 66, 80, and 128.
- You can add a bypass interface to register and output the input data through a bypass port using the following parameter:

  Enable Bypass Interface — Select this option to enable this interface. By default, this interface is disabled.
- The data pattern checker core supports two types of interface: Avalon-ST and Conduit interface. You can select either of them using the following parameter:

  Enable Avalon Interface — Select this option to enable Avalon interface. By default this interface is enabled. De-select this option to enable Conduit interface.
- You can enable frequency counter by selecting the following parameter:

  Enable Frequency Counter
- The following parameter determines the synchronization depth for clock crossing from Avalon-MM clock domain to Avalon-ST clock domain:

  CROSS_CLK_SYNC_DEPTH — Default value is 2. Valid values are => 2.

38.4. Hardware Simulation Considerations

The data pattern generator and checker cores do not provide a simulation testbench for simulating a stand-alone instance of the component. However, you can use the standard Platform Designer simulation flow to simulate the component design files inside an Platform Designer system.

38.5. Software Programming Model

This section describes the software programming model for the data pattern generator and checker cores.

38.5.1. Register Maps

This section describes the register maps for the data pattern generator and checker cores.

Data Pattern Generator Control Registers

Table 389. Data Pattern Generator Register Map

Offset	Register Name
base + 0	`Enable`
base + 1	`Pattern Select`
base + 2	`Inject Error`
base + 3	Preamble Control
base + 4	Preamble Character (Lower Bits)
base + 5	Preamble Character (Higher Bits)

Table 390. Enable Field Descriptions

Bit(s)	Name	Access	Description
[0]	EN	RW	Setting this bit to 1 enables the data pattern generator core.
[31:1]	Reserved		

Note to **Table 29–4** :

1. When the core is enabled, only the `Enable` register and the `Inject Error` register have write access. Write access to all other registers are ignored.The first valid data is observed from the Avalon-ST Source interface at the fourth cycle after the `Enable` bit is set. When the core is disabled, the final output is observed at the next clock cycle.

Table 391. Pattern Select Field Descriptions

Bit(s)	Name	Access	Description
[0]	PRBS7	RW	Setting this bit to 1 outputs a PRBS 7 pattern with T [7, 6].
[1]	PRBS15	RW	Setting this bit to 1 outputs a PRBS 15 pattern with T [15, 14].
[2]	PRBS23	RW	Setting this bit to 1 outputs a PRBS 23 pattern with T [23, 18].
[3]	PRBS31	RW	Setting this bit to 1 outputs a PRBS 31 pattern with T [31, 28].
[4]	HF	RW	Setting this bit to 1 outputs a constant pattern of 0101010101… bits.

continued...

Bit(s)	Name	Access	Description
[5]	LF	RW	Setting this bit to 1 outputs a constant word pattern of 1111100000 for 10-bit words, or 11110000 for 8-bit words.
[31:8]	Reserved		

Note to **Table 29–5** :
1. This register is one-hot encoded where only one of the pattern selector bits should be set to 1. For all other settings, the behaviors are undefined.

This register allows you to set the error inject bit and insert one bit of error into the stream.

Table 392. Inject Error Field Descriptions (Note 1)

Bit(s)	Name	Access	Description
[0]	IJ	RW	Setting this bit to 1 injects error into the stream. If the IJ bit is set to 1 when the core is enabled, the bit resets itself to 0 at the next clock cycle when the error is injected.
[31:1]	Reserved		

Note to **Table 29–6** :
1. The LSB of the data beat is flipped at the fourth clock cycle after the IJ bit is set (if not being backpressured by the sink when it is valid). The data beat that is injected with error might not be observed from the source if the core is disabled within the next two cycles after IJ bit is set to 1.

This register enables preamble and set the number of cycles to output the preamble character.

Table 393. Preamble Control Field Descriptions

Bit(s)	Name	Access	Description
[0]	EP	RW	Setting this bit to 1, at the start of pattern generation, enables the preamble character to be sent for Numbits cycles before switching over to the selected pattern.
[7:1]	Reserved		
[15:8]	Numbits	RW	The number of bits to repeat the preamble character.
[31:16]	Reserved		

This register is for the user-defined preamble character (bit 0-31).

Table 394. Preamble Character Low Bits Field Descriptions

Bit(s)	Name	Access	Description
[31:0]	Preamble Character (Lower Bits)	RW	Sets bit 31-0 for the preamble character to output.

This register is for the user-defined preamble character (bit 32-39) but is ignored if the ST_DATA_W value is set to 32.

Table 395. Preamble Character High Bits Field Descriptions

Bit(s)	Name	Access	Description
[7:0]	Preamble Character (Higher Bits)	RW	Sets bit 39-32 for the preamble character. This is ignored when the ST_DATA_W value is set to 32.
[31:8]	Reserved		

Data Pattern Checker Control and Status Registers

Table 396. Data Pattern Checker Control and Status Register Map

Offset	Register Name
base + 0	Status
base + 1	Pattern Set
base + 2	Counter Control
base + 3	NumBits (Lower Bits)
base + 4	NumBits (Higher Bits)
base + 5	NumErrors (Lower Bits)
base + 6	NumErrors (Higher Bits)

Table 397. Status Field Descriptions

Bit(s)	Name	Access	Description
[0]	EN	RW	Setting this bit to 1 enables pattern checking.
[1]	LK	R	Indicate lock status (writing to this bit has no effect).
[31:2]	Reserved		

Note to **Table 29–11** :

1. When the core is enabled, only the Status register's EN bit and the counter control register have write access. Write access to all other registers are ignored.

Table 398. Pattern Select Field Descriptions

Bit(s)	Name	Access	Description
[0]	PRBS7	RW	Setting this bit to 1 compares the data to a PRBS 7 pattern with T [7, 6].
[1]	PRBS15	RW	Setting this bit to 1 compares the data to a PRBS 15 pattern with T [15, 14].
[2]	PRBS23	RW	Setting this bit to 1 compares the data to a PRBS 23 pattern with T [23, 18].
[3]	PRBS31	RW	Setting this bit to 1 compares the data to a PRBS 31 pattern with T [31, 28].
[4]	HF	RW	Setting this bit to 1 compares the data to a constant pattern of 0101010101… bits.
[5]	LF	RW	Setting this bit to 1 compares the data to a constant word pattern of 1111100000 for 10-bit words, or 11110000 for 8-bit words.
[31:8]	Reserved		

Note to **Table 29–12** :

continued...

Bit(s)	Name	Access	Description
1. This register is one-hot encoded where only one of the pattern selector bits should be set to 1. For all other settings, the behaviors are undefined.			

This register snapshots and resets the NumBits, NumErrors, and also the internal counters.

Table 399. Counter Control Field Descriptions

Bit(s)	Name	Access	Description
[0]	SN	W	Writing this bit to 1 captures the number of bits received and number of error bits received from the internal counters to the respective NumBits and NumErrors registers within the same clock cycle. Writing this bit to 1 after disabling the core will still capture the correct values from the internal counters to the NumBits and NumErrors registers.
[1]	RST	W	Writing this bit to 1 resets all internal counters and statistics. This bit resets itself automatically after the reset process. Re-enabling the core does not automatically reset the number of bits received and number of error bits received in the internal counter.
[31:2]	Reserved		

This register is the lower word of the 64-bit bit counter snapshot value. The register is reset when the component-reset is asserted or when the RST bit is set to 1.

Table 400. NumBits (Lower Word) Field Descriptions

Bit(s)	Name	Access	Description
[31:0]	`NumBits (Lower Bits)`	R	Sets bit 31-0 for the `NumBits` (number of bits received).

This register is the higher word of the 64-bit bit counter snapshot value. The register is reset when the component-reset is asserted or when the RST bit is set to 1.

Table 401. NumBits (Higher Word) Field Descriptions

Bit(s)	Name	Access	Description
[31:0]	`NumBits (Higher Bits)`	R	Sets bit 63-32 for the `NumBits` (number of bits received).

This register is the lower word of the 64-bit error counter snapshot value. The register is reset when the component-reset is asserted or when the RST bit is set to 1.

Table 402. NumErrors (Lower Word) Field Descriptions

Bit(s)	Name	Access	Description
[31:0]	`NumErrors (Lower Bits)`	R	Sets bit 31-0 for the `NumErrors` (number of error bits received).

This register is the higher word of the 64-bit error counter snapshot value. The register is reset when the component-reset is asserted or when the RST bit is set to 1.

Table 403. NumErrors (Higher Word) Field Descriptions

Bit(s)	Name	Access	Description
[31:0]	NumErrors (Higher Bits)	R	Sets bit 63-32 for the NumErrors (number of error bits received).

38.6. Avalon-ST Data Pattern Generator and Checker Cores Revision History

Document Version	Intel Quartus Prime Version	Changes
2020.09.21	20.2	Corrected *Table: Counter Control Field Descriptions*.
2018.05.07	18.0	Corrected *Table: Counter Control Field Descriptions*.

Date	Version	Changes
November 2017	2017.11.06	Updated the configuration information for both Data Pattern Generator and Checker.
December 2010	v10.1.0	Removed the "Device Support", "Instantiating the Core in SOPC Builder", and "Referenced Documents" sections.
July 2010	v10.0.0	No change from previous release.
January 2010	v9.1.1	Initial release.

39. Avalon-ST Test Pattern Generator and Checker Cores

39.1. Core Overview

The data generation and monitoring solution for Avalon Streaming (Avalon-ST) consists of two components: a test pattern generator core that generates packetized or non-packetized data and sends it out on an Avalon-ST data interface, and a test pattern checker core that receives the same data and checks it for correctness.

The test pattern generator core can insert different error conditions, and the test pattern checker reports these error conditions to the control interface, each via an Avalon Memory-Mapped (Avalon-MM) agent.

39.2. Resource Utilization and Performance

Resource utilization and performance for the test pattern generator and checker cores depend on the data width, number of channels, and whether the streaming data uses the optional packet protocol.

Table 404. Test Pattern Generator Estimated Resource Utilization and Performance

No. of Channels	Datawidth (No. of 8-bit Symbols Per Beat)	Packet Support	Stratix II and Stratix II GX			Cyclone II			Stratix		
			f_{MAX} (MHz)	ALM Count	Memory (bits)	f_{MAX} (MHz)	Logic Cells	Memory (bits)	f_{MAX} (MHz)	Logic Cells	Memory (bits)
1	4	Yes	284	233	560	206	642	560	202	642	560
1	4	No	293	222	496	207	572	496	245	561	496
32	4	Yes	276	270	912	210	683	912	197	707	912
32	4	No	323	227	848	234	585	848	220	630	848
1	16	Yes	298	361	560	228	867	560	245	896	560
1	16	No	340	330	496	230	810	496	228	845	496
32	16	Yes	295	410	912	209	954	912	224	956	912
32	16	No	269	409	848	219	842	848	204	912	848

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Table 405. Test Pattern Checker Estimated Resource Utilization and Performance

No. of Channels	Datawidth (No. of 8-bit Symbols Per Beat)	Packet Support	Stratix II and Stratix II GX			Cyclone II			Stratix		
			f_{MAX} (MHz)	ALM Count	Memory (bits)	f_{MAX} (MHz)	Logic Cells	Memory (bits)	f_{MAX} (MHz)	Logic Cells	Memory (bits)
1	4	Yes	270	271	96	179	940	0	174	744	96
1	4	No	371	187	32	227	628	0	229	663	32
32	4	Yes	185	396	3616	111	875	3854	105	795	3616
32	4	No	221	363	3520	133	686	3520	133	660	3520
1	16	Yes	253	462	96	185	1433	0	166	1323	96
1	16	No	277	306	32	218	1044	0	192	1004	32
32	16	Yes	182	582	3616	111	1367	3584	110	1298	3616
32	16	No	218	473	3520	129	1143	3520	126	1074	3520

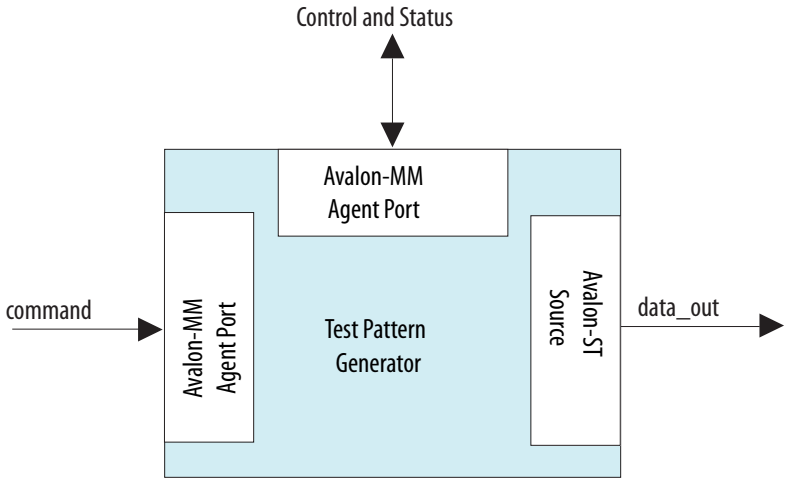
39.3. Test Pattern Generator

This section describes the hardware structure and functionality of the test pattern generator core.

39.3.1. Functional Description

The test pattern generator core accepts commands to generate data via an Avalon-MM command interface, and drives the generated data to an Avalon-ST data interface. You can parameterize most aspects of the Avalon-ST data interface such as the number of error bits and data signal width, thus allowing you to test components with different interfaces.

Test Pattern Generator Core Block Diagram



The data pattern is determined by the following equation:
Symbol Value = Symbol Position in Packet X0R Data Error Mask. Non-packetized data is one long stream with no beginning or end.

The test pattern generator core has a throttle register that is set via the Avalon-MM control interface. The value of the throttle register is used in conjunction with a pseudo-random number generator to throttle the data generation rate.

Command Interface

The command interface is a 32-bit Avalon-MM write agent that accepts data generation commands. It is connected to a 16-element deep FIFO, thus allowing a host peripheral to drive a number of commands into the test pattern generator core.

The command interface maps to the following registers: `cmd_lo` and `cmd_hi`. The command is pushed into the FIFO when the register `cmd_lo` (address 0) is written to. When the FIFO is full, the command interface asserts the `waitrequest` signal. You can create errors by writing to the register `cmd_hi` (address 1). The errors are only cleared when 0 is written to this register or its respective fields. See page the **Test Pattern Generator Command Registers** section for more information on the register fields.

Control and Status Interface

The control and status interface is a 32-bit Avalon-MM agent that allows you to enable or disable the data generation as well as set the throttle.

This interface also provides useful generation-time information such as the number of channels and whether or not packets are supported.

Output Interface

The output interface is an Avalon-ST interface that optionally supports packets. You can configure the output interface to suit your requirements.

Depending on the incoming stream of commands, the output data may contain interleaved packet fragments for different channels. To keep track of the current symbol's position within each packet, the test pattern generator core maintains an internal state for each channel.

39.3.2. Configuration

The following sections list the available options in the MegaWizard™ interface.

Functional Parameter

The functional parameter allows you to configure the test pattern generator as a whole: **Throttle Seed**—The starting value for the throttle control random number generator. Intel recommends a value which is unique to each instance of the test pattern generator and checker cores in a system.

Output Interface

You can configure the output interface of the test pattern generator core using the following parameters:

- **Number of Channels**—The number of channels that the test pattern generator core supports. Valid values are 1 to 256.
- **Data Bits Per Symbol**—The number of bits per symbol for the input and output interfaces. Valid values are 1 to 256. Example—For typical systems that carry 8-bit bytes, set this parameter to 8.
- **Data Symbols Per Beat**—The number of symbols (words) that are transferred per beat. Valid values are 1 to 256.
- **Include Packet Support**—Indicates whether or not packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
- **Error Signal Width (bits)**—The width of the `error` signal on the output interface. Valid values are 0 to 31. A value of 0 indicates that the `error` signal is not used.

39.4. Test Pattern Checker

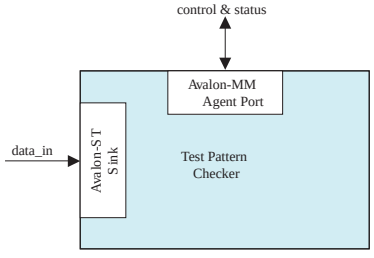
This section describes the hardware structure and functionality of the test pattern checker core.

39.4.1. Functional Description

The test pattern checker core accepts data via an Avalon-ST interface, checks it for correctness against the same predetermined pattern used by the test pattern generator core to produce the data, and reports any exceptions to the control interface. You can parameterize most aspects of the test pattern checker's Avalon-ST interface such as the number of error bits and the data signal width, thus allowing you to test components with different interfaces.

The test pattern checker has a throttle register that is set via the Avalon-MM control interface. The value of the throttle register controls the rate at which data is accepted.

Figure 125. Test Pattern Checker



The test pattern checker core detects exceptions and reports them to the control interface via a 32-element deep internal FIFO. Possible exceptions are data error, missing start-of-packet (SOP), missing end-of-packet (EOP) and signalled error.

As each exception occurs, an exception descriptor is pushed into the FIFO. If the same exception occurs more than once consecutively, only one exception descriptor is pushed into the FIFO. All exceptions are ignored when the FIFO is full. Exception descriptors are deleted from the FIFO after they are read by the control and status interface.

Input Interface

The input interface is an Avalon-ST interface that optionally supports packets. You can configure the input interface to suit your requirements.

Incoming data may contain interleaved packet fragments. To keep track of the current symbol's position, the test pattern checker core maintains an internal state for each channel.

Control and Status Interface

The control and status interface is a 32-bit Avalon-MM agent that allows you to enable or disable data acceptance as well as set the throttle. This interface provides useful generation-time information such as the number of channels and whether the test pattern checker supports packets.

The control and status interface also provides information on the exceptions detected by the test pattern checker core. The interface obtains this information by reading from the exception FIFO.

39.4.2. Configuration

The following sections list the available options in the MegaWizard™ interface.

Functional Parameter

The functional parameter allows you to configure the test pattern checker as a whole:
Throttle Seed—The starting value for the throttle control random number generator. Intel recommends a unique value to each instance of the test pattern generator and checker cores in a system.

Input Parameters

You can configure the input interface of the test pattern checker core using the following parameters:

- **Data Bits Per Symbol**—The number of bits per symbol for the input interface. Valid values are 1 to 256.
- **Data Symbols Per Beat**—The number of symbols (words) that are transferred per beat. Valid values are 1 to 32.
- **Include Packet Support**—Indicates whether or not packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
- **Number of Channels**—The number of channels that the test pattern checker core supports. Valid values are 1 to 256.
- **Error Signal Width (bits)**—The width of the `error` signal on the input interface. Valid values are 0 to 31. A value of 0 indicates that the `error` signal is not in use.

39.5. Hardware Simulation Considerations

The test pattern generator and checker cores do not provide a simulation testbench for simulating a stand-alone instance of the component. However, you can use the standard Platform Designer simulation flow to simulate the component design files inside an Platform Designer system.

39.6. Software Programming Model

This section describes the software programming model for the test pattern generator and checker cores.

39.6.1. HAL System Library Support

For Nios II processor users, Intel provides HAL system library drivers that enable you to initialize and access the test pattern generator and checker cores. Intel recommends you to use the provided drivers to access the cores instead of accessing the registers directly.

For Nios II IDE users, copy the provided drivers from the following installation folders to your software application directory:

- <IP installation directory> **/ip /sopc_builder_ip / altera_avalon_data_source/HAL**
- <IP installation directory>**/ip/sopc_builder_ip/ altera_avalon_data_sink/HAL**

  This instruction does not apply if you use the Nios II command-line tools.

39.6.2. Software Files

The following software files define the low-level access to the hardware, and provide the routines for the HAL device drivers. Application developers should not modify these files.

- Software files provided with the test pattern generator core:
  - `data_source_regs.h`—The header file that defines the test pattern generator's register maps.
  - `data_source_util.h, data_source_util.c`—The header and source code for the functions and variables required to integrate the driver into the HAL system library.
- Software files provided with the test pattern checker core:
  - `data_sink_regs.h`—The header file that defines the core's register maps.
  - `data_sink_util.h, data_sink_util.c`—The header and source code for the functions and variables required to integrate the driver into the HAL system library.

39.6.3. Register Maps

This section describes the register maps for the test pattern generator and checker cores.

Test Pattern Generator Control and Status Registers

The table below shows the offset for the test pattern generator control and status registers. Each register is 32 bits wide.

Table 406. Test Pattern Generator Control and Status Register Map

Offset	Register Name
base + 0	status
base + 1	control
base + 2	fill

Table 407. Status Field Descriptions

Bit(s)	Name	Access	Description
[15:0]	ID	RO	A constant value of 0x64.
[23:16]	NUMCHANNELS	RO	The configured number of channels.
[30:24]	NUMSYMBOLS	RO	The configured number of symbols per beat.
[31]	SUPPORTPACKETS	RO	A value of 1 indicates packet support.

Table 408. Control Field Descriptions

Bit(s)	Name	Access	Description
[0]	ENABLE	RW	Setting this bit to 1 enables the test pattern generator core.
[7:1]	Reserved		
[16:8]	THROTTLE	RW	Specifies the throttle value which can be between 0–256, inclusively. This value is used in conjunction with a pseudorandom number generator to throttle the data generation rate. Setting THROTTLE to 0 stops the test pattern generator core. Setting it to 256 causes the test pattern generator core to run at full throttle. Values between 0–256 result in a data rate proportional to the throttle value.
[17]	SOFT RESET	RW	When this bit is set to 1, all internal counters and statistics are reset. Write 0 to this bit to exit reset.
[31:18]	Reserved		

Table 409. Fill Field Descriptions

Bit(s)	Name	Access	Description
[0]	BUSY	RO	A value of 1 indicates that data transmission is in progress, or that there is at least one command in the command queue.
[6:1]	Reserved		
[15:7]	FILL	RO	The number of commands currently in the command FIFO.
[31:16]	Reserved		

Test Pattern Generator Command Registers

The table below shows the offset for the command registers. Each register is 32 bits wide.

Table 410. Test Pattern Command Register Map

Offset	Register Name
base + 0	cmd_lo
base + 1	cmd_hi

The command is pushed into the FIFO only when the cmd_lo register is written to.

Table 411. cmd_lo Field Descriptions

Bit(s)	Name	Access	Description
[15:0]	SIZE	RW	The segment size in symbols. Except for the last segment in a packet, the size of all segments must be a multiple of the configured number of symbols per beat. If this condition is not met, the test pattern generator core inserts additional symbols to the segment to ensure the condition is fulfilled.
[29:16]	CHANNEL	RW	The channel to send the segment on. If the channel signal is less than 14 bits wide, the low order bits of this register are used to drive the signal.
[30]	SOP	RW	Set this bit to 1 when sending the first segment in a packet. This bit is ignored when packets are not supported.
[31]	EOP	RW	Set this bit to 1 when sending the last segment in a packet. This bit is ignored when packets are not supported.

Table 412. cmd_hi Field Descriptions

Bit(s)	Name	Access	Description
[15:0]	SIGNALLED ERROR	RW	Specifies the value to drive the error signal. A non-zero value creates a signalled error.
[23:16]	DATA ERROR	RW	The output data is XORed with the contents of this register to create data errors. To stop creating data errors, set this register to 0.
[24]	SUPRESS SOP	RW	Set this bit to 1 to suppress the assertion of the startofpacket signal when the first segment in a packet is sent.
[25]	SUPRESS EOP	RW	Set this bit to 1 to suppress the assertion of the endofpacket signal when the last segment in a packet is sent.

Test Pattern Checker Control and Status Registers

The table below shows the offset for the control and status registers. Each register is 32 bits wide.

Table 413. Test Pattern Checker Control and Status Register Map

Offset	Register Name
base + 0	status
base + 1	control
base + 2	Reserved
base + 3	
base + 4	

continued...

Offset	Register Name
base + 5	exception_descriptor
base + 6	indirect_select
base + 7	indirect_count

Table 414. Status Field Descriptions

Bit(s)	Name	Access	Description
[15:0]	ID	RO	Contains a constant value of 0x65.
[23:16]	NUMCHANNELS	RO	The configured number of channels.
[30:24]	NUMSYMBOLS	RO	The configured number of symbols per beat.
[31]	SUPPORTPACKETS	RO	A value of 1 indicates packet support.

Table 415. Control Field Descriptions

Bit(s)	Name	Access	Description
[0]	ENABLE	RW	Setting this bit to 1 enables the test pattern checker.
[7:1]	Reserved		
[16:8]	THROTTLE	RW	Specifies the throttle value which can be between 0–256, inclusively. This value is used in conjunction with a pseudorandom number generator to throttle the data generation rate. Setting THROTTLE to 0 stops the test pattern generator core. Setting it to 256 causes the test pattern generator core to run at full throttle. Values between 0–256 result in a data rate proportional to the throttle value.
[17]	SOFT RESET	RW	When this bit is set to 1, all internal counters and statistics are reset. Write 0 to this bit to exit reset.
[31:18]	Reserved		

The table below describes the exception_descriptor register bits. If there is no exception, reading this register returns 0.

Table 416. exception_descriptor Field Descriptions

Bit(s)	Name	Access	Description
[0]	DATA ERROR	RO	A value of 1 indicates that an error is detected in the incoming data.
[1]	MISSINGSOP	RO	A value of 1 indicates missing start-of-packet.
[2]	MISSINGEOP	RO	A value of 1 indicates missing end-of-packet.
[7:3]	Reserved		
[15:8]	SIGNALLED ERROR	RO	The value of the error signal.
[23:16]	Reserved		
[31:24]	CHANNEL	RO	The channel on which the exception was detected.

Table 417. indirect_select Field Descriptions

Bit	Bits Name	Access	Description
[7:0]	INDIRECT CHANNEL	RW	Specifies the channel number that applies to the INDIRECT PACKET COUNT, INDIRECT SYMBOL COUNT, and INDIRECT ERROR COUNT registers.
[15:8]	Reserved		
[31:16]	INDIRECT ERROR	RO	The number of data errors that occurred on the channel specified by INDIRECT CHANNEL.

Table 418. indirect_count Field Descriptions

Bit	Bits Name	Access	Description
[15:0]	INDIRECT PACKET COUNT	RO	The number of packets received on the channel specified by INDIRECT CHANNEL.
[31:16]	INDIRECT SYMBOL COUNT	RO	The number of symbols received on the channel specified by INDIRECT CHANNEL.

39.7. Test Pattern Generator API

This section describes the application programming interface (API) for the test pattern generator core. All API functions are currently not available from the interrupt service routine (ISR).

39.7.1. data_source_reset()

Prototype:	`void data_source_reset(alt_u32 base);`
Thread-safe:	No.
Include:	**<data_source_util.h>**
Parameters:	`base`—The base address of the control and status agent.
Returns:	`void`.
Description:	This function resets the test pattern generator core including all internal counters and FIFOs. The control and status registers are not reset by this function.

39.7.2. data_source_init()

Prototype:	`int data_source_init(alt_u32 base, alt_u32 command_base);`
Thread-safe:	No.
Include:	**<data_source_util.h>**
Parameters:	`base`—The base address of the control and status agent. `command_base`—The base address of the command agent.
Returns:	1—Initialization is successful. 0—Initialization is unsuccessful.
Description:	This function performs the following operations to initialize the test pattern generator core: Resets and disables the test pattern generator core. Sets the maximum throttle. Clears all inserted errors.

39.7.3. data_source_get_id()

Prototype:	`int data_source_get_id(alt_u32 base);`
Thread-safe:	Yes.
Include:	**<data_source_util.h>**
Parameters:	`base`—The base address of the control and status agent.
Returns:	The test pattern generator core's identifier.
Description:	This function retrieves the test pattern generator core's identifier.

39.7.4. data_source_get_supports_packets()

Prototype:	`int data_source_init(alt_u32 base);`
Thread-safe:	Yes.
Include:	**<data_source_util.h>**
Parameters:	`base`—The base address of the control and status agent.
Returns:	1—Packets are supported. 0—Packets are not supported.
Description:	This function checks if the test pattern generator core supports packets.

39.7.5. data_source_get_num_channels()

Prototype:	`int data_source_get_num_channels(alt_u32 base);`
Thread-safe:	Yes.
Include:	**<data_source_util.h>**
Parameters:	`base`—The base address of the control and status agent.
Returns:	The number of channels supported.
Description:	This function retrieves the number of channels supported by the test pattern generator core.

39.7.6. data_source_get_symbols_per_cycle()

Prototype:	`int data_source_get_symbols(alt_u32 base);`
Thread-safe:	Yes.
Include:	**<data_source_util.h>**
Parameters:	`base`—The base address of the control and status agent.
Returns:	The number of symbols transferred in a beat.
Description:	This function retrieves the number of symbols transferred by the test pattern generator core in each beat.

39.7.7. data_source_set_enable()

Prototype:	`void data_source_set_enable(alt_u32 base, alt_u32 value);`
Thread-safe:	No.
Include:	**<data_source_util.h>**
Parameters:	`base`—The base address of the control and status agent. `value`—The ENABLE bit is set to the value of this parameter.
Returns:	`void`.
Description:	This function enables or disables the test pattern generator core. When disabled, the test pattern generator core stops data transmission but continues to accept commands and stores them in the FIFO.

39.7.8. data_source_get_enable()

Prototype:	`int data_source_get_enable(alt_u32 base);`
Thread-safe:	Yes.
Include:	**<data_source_util.h>**
Parameters:	`base`—The base address of the control and status agent.
Returns:	The value of the ENABLE bit.
Description:	This function retrieves the value of the ENABLE bit.

39.7.9. data_source_set_throttle()

Prototype:	`void data_source_set_throttle(alt_u32 base, alt_u32 value);`
Thread-safe:	No.
Include:	**<data_source_util.h>**
Parameters:	`base`—The base address of the control and status agent. `value`—The throttle value.
Returns:	`void`.
Description:	This function sets the throttle value, which can be between 0–256 inclusively. The throttle value, when divided by 256 yields the rate at which the test pattern generator sends data.

39.7.10. data_source_get_throttle()

Prototype:	`int data_source_get_throttle(alt_u32 base);`
Thread-safe:	Yes.
Include:	**<data_source_util.h>**
Parameters:	`base`—The base address of the control and status agent.
Returns:	The throttle value.
Description:	This function retrieves the current throttle value.

39.7.11. data_source_is_busy()

Prototype:	`int data_source_is_busy(alt_u32 base);`
Thread-safe:	Yes.
Include:	**<data_source_util.h>**
Parameters:	`base`—The base address of the control and status agent.
Returns:	1—The test pattern generator core is busy. 0—The core is not busy.
Description:	This function checks if the test pattern generator is busy. The test pattern generator core is busy when it is sending data or has data in the command FIFO to be sent.

39.7.12. data_source_fill_level()

Prototype:	`int data_source_fill_level(alt_u32 base);`
Thread-safe:	Yes.
Include:	**<data_source_util.h>**
Parameters:	`base`—The base address of the control and status agent.
Returns:	The number of commands in the command FIFO.
Description:	This function retrieves the number of commands currently in the command FIFO.

39.7.13. data_source_send_data()

Prototype:	`int data_source_send_data(alt_u32 cmd_base, alt_u16 channel, alt_u16 size, alt_u32 flags, alt_u16 error, alt_u8 data_error_mask);`
Thread-safe:	No.
Include:	**<data_source_util.h>**
Parameters:	`cmd_base`—The base address of the command agent. `channel`—The channel to send the data on. `size`—The data size. `flags`—Specifies whether to send or suppress SOP and EOP signals. Valid values are `DATA_SOURCE_SEND_SOP`, `DATA_SOURCE_SEND_EOP`, `DATA_SOURCE_SEND_SUPRESS_SOP` and `DATA_SOURCE_SEND_SUPRESS_EOP`. `error`—The value asserted on the `error` signal on the output interface. `data_error_mask`—This parameter and the data are XORed together to produce erroneous data.
Returns:	Always returns 1.
Description:	This function sends a data fragment to the specified channel. If packets are supported, user applications must ensure the following conditions are met: SOP and EOP are used consistently in each channel. Except for the last segment in a packet, the length of each segment is a multiple of the data width. If packets are not supported, user applications must ensure the following conditions are met: No SOP and EOP indicators in the data. The length of each segment in a packet is a multiple of the data width.

39.8. Test Pattern Checker API

This section describes the API for the test pattern checker core. The API functions are currently not available from the ISR.

39.8.1. data_sink_reset()

Prototype:	`void data_sink_reset(alt_u32 base);`
Thread-safe:	No.
Include:	**<data_sink_util.h>**
Parameters:	`base`—The base address of the control and status agent.
Returns:	`void`.
Description:	This function resets the test pattern checker core including all internal counters.

39.8.2. data_sink_init()

Prototype:	`int data_source_init(alt_u32 base);`
Thread-safe:	No.
Include:	**<data_sink_util.h>**
Parameters:	`base`—The base address of the control and status agent.
Returns:	1—Initialization is successful. 0—Initialization is unsuccessful.
Description:	This function performs the following operations to initialize the test pattern checker core: Resets and disables the test pattern checker core. Sets the throttle to the maximum value.

39.8.3. data_sink_get_id()

Prototype:	`int data_sink_get_id(alt_u32 base);`
Thread-safe:	Yes.
Include:	**<data_sink_util.h>**
Parameters:	`base`—The base address of the control and status agent.
Returns:	The test pattern checker core's identifier.
Description:	This function retrieves the test pattern checker core's identifier.

39.8.4. data_sink_get_supports_packets()

Prototype:	`int data_sink_init(alt_u32 base);`
Thread-safe:	Yes.
Include:	**<data_sink_util.h>**
Parameters:	`base`—The base address of the control and status agent.
Returns:	1—Packets are supported.

continued...

	0—Packets are not supported.
Description:	This function checks if the test pattern checker core supports packets.

39.8.5. data_sink_get_num_channels()

Prototype:	`int data_sink_get_num_channels(alt_u32 base);`
Thread-safe:	Yes.
Include:	**<data_sink_util.h>**
Parameters:	`base`—The base address of the control and status agent.
Returns:	The number of channels supported.
Description:	This function retrieves the number of channels supported by the test pattern checker core.

39.8.6. data_sink_get_symbols_per_cycle()

Prototype:	`int data_sink_get_symbols(alt_u32 base);`
Thread-safe:	Yes.
Include:	**<data_sink_util.h>**
Parameters:	`base`—The base address of the control and status agent.
Returns:	The number of symbols received in a beat.
Description:	This function retrieves the number of symbols received by the test pattern checker core in each beat.

39.8.7. data_sink_set enable()

Prototype:	`void data_sink_set_enable(alt_u32 base, alt_u32 value);`
Thread-safe:	No.
Include:	**<data_sink_util.h>**
Parameters:	`base`—The base address of the control and status agent. `value`—The ENABLE bit is set to the value of this parameter.
Returns:	`void`.
Description:	This function enables the test pattern checker core.

39.8.8. data_sink_get_enable()

Prototype:	`int data_sink_get_enable(alt_u32 base);`
Thread-safe:	Yes.
Include:	**<data_sink_util.h>**
Parameters:	`base`—The base address of the control and status agent.
Returns:	The value of the ENABLE bit.
Description:	This function retrieves the value of the ENABLE bit.

39.8.9. data_sink_set_throttle()

Prototype:	`void data_sink_set_throttle(alt_u32 base, alt_u32 value);`
Thread-safe:	No.
Include:	**<data_sink_util.h>**
Parameters:	`base`—The base address of the control and status agent. `value`—The throttle value.
Returns:	`void`.
Description:	This function sets the throttle value, which can be between 0–256 inclusively. The throttle value, when divided by 256 yields the rate at which the test pattern checker receives data.

39.8.10. data_sink_get_throttle()

Prototype:	`int data_sink_get_throttle(alt_u32 base);`
Thread-safe:	Yes.
Include:	**<data_sink_util.h>**
Parameters:	`base`—The base address of the control and status agent.
Returns:	The throttle value.
Description:	This function retrieves the throttle value.

39.8.11. data_sink_get_packet_count()

Prototype:	`int data_sink_get_packet_count(alt_u32 base, alt_u32 channel);`
Thread-safe:	No.
Include:	**<data_sink_util.h>**
Parameters:	`base`—The base address of the control and status agent. `channel`—Channel number.
Returns:	The number of packets received on the given channel.
Description:	This function retrieves the number of packets received on a given channel.

39.8.12. data_sink_get_symbol_count()

Prototype:	`int data_sink_get_symbol_count(alt_u32 base, alt_u32 channel);`
Thread-safe:	No.
Include:	**<data_sink_util.h>**
Parameters:	`base`—The base address of the control and status agent. `channel`—Channel number.
Returns:	The number of symbols received on the given channel.
Description:	This function retrieves the number of symbols received on a given channel.

39.8.13. data_sink_get_error_count()

Prototype:	`int data_sink_get_error_count(alt_u32 base, alt_u32 channel);`
Thread-safe:	No.
Include:	<**data_sink_util.h**>
Parameters:	`base`—The base address of the control and status agent. `channel`—Channel number.
Returns:	The number of errors received on the given channel.
Description:	This function retrieves the number of errors received on a given channel.

39.8.14. data_sink_get_exception()

Prototype:	`int data_sink_get_exception(alt_u32 base);`
Thread-safe:	Yes.
Include:	**<data_sink_util.h>**
Parameters:	`base`—The base address of the control and status agent.
Returns:	The first exception descriptor in the exception FIFO. 0—No exception descriptor found in the exception FIFO.
Description:	This function retrieves the first exception descriptor in the exception FIFO and pops it off the FIFO.

39.8.15. data_sink_exception_is_exception()

Prototype:	`int data_sink_exception_is_exception(int exception);`
Thread-safe:	Yes.
Include:	**<data_sink_util.h>**
Parameters:	`exception`—Exception descriptor
Returns:	1—Indicates an exception. 0—No exception.
Description:	This function checks if a given exception descriptor describes a valid exception.

39.8.16. data_sink_exception_has_data_error()

Prototype:	`int data_sink_exception_has_data_error(int exception);`
Thread-safe:	Yes.
Include:	**<data_sink_util.h>**
Parameters:	`exception`—Exception descriptor.
Returns:	1—Data has errors. 0—No errors.
Description:	This function checks if a given exception indicates erroneous data.

39.8.17. data_sink_exception_has_missing_sop()

Prototype:	`int data_sink_exception_has_missing_sop(int exception);`
Thread-safe:	Yes.
Include:	**<data_sink_util.h>**
Parameters:	`exception`—Exception descriptor.
Returns:	1—Missing SOP. 0—Other exception types.
Description:	This function checks if a given exception descriptor indicates missing SOP.

39.8.18. data_sink_exception_has_missing_eop()

Prototype:	`int data_sink_exception_has_missing_eop(int exception);`
Thread-safe:	Yes.
Include:	**<data_sink_util.h>**
Parameters:	`exception`—Exception descriptor.
Returns:	1—Missing EOP. 0—Other exception types.
Description:	This function checks if a given exception descriptor indicates missing EOP.

39.8.19. data_sink_exception_signalled_error()

Prototype:	`int data_sink_exception_signalled_error(int exception);`
Thread-safe:	Yes.
Include:	**<data_sink_util.h>**
Parameters:	`exception`—Exception descriptor.
Returns:	The signalled error value.
Description:	This function retrieves the value of the signalled error from the exception.

39.8.20. data_sink_exception_channel()

Prototype:	`int data_sink_exception_channel(int exception);`
Thread-safe:	Yes.
Include:	**<data_sink_util.h>**
Parameters:	`exception`—Exception descriptor.
Returns:	The channel number on which the given exception occurred.
Description:	This function retrieves the channel number on which a given exception occurred.

39.9. Avalon-ST Test Pattern Generator and Checker Cores Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
December 2010	v10.1.0	Removed the "Device Support", "Instantiating the Core in SOPC Builder", and "Referenced Documents" sections.
July 2010	v10.0.0	No change from previous release.
November 2009	v9.1.0	No change from previous release.
March 2009	v9.0.0	No change from previous release.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	v8.0.0	Updated the section on HAL System Library Support.

40. System ID Peripheral Core

40.1. Core Overview

The system ID core with Avalon interface is a simple read-only device that provides Platform Designer systems with a unique identifier. Nios II processor systems use the system ID core to verify that an executable program was compiled targeting the actual hardware image configured in the target FPGA. If the expected ID in the executable does not match the system ID core in the FPGA, it is possible that the software will not execute correctly.

40.2. Functional Description

The system ID core provides a read-only Avalon Memory-Mapped (Avalon-MM) agent interface. This interface has two 32-bit registers, as shown in the table below. The value of each register is determined at system generation time, and always returns a constant value.

Table 419. System ID Core Register Map

Offset	Register Name	R/W	Description
0	`id`	R	A unique 32-bit value that is based on the contents of the Platform Designer system. The id is similar to a check-sum value; Platform Designer systems with different components, different configuration options, or both, produce different id values.
1	`timestamp`	R	A unique 32-bit value that is based on the system generation time. The value is equivalent to the number of seconds after Jan. 1, 1970.

There are two basic ways to use the system ID core:

- Verify the system ID before downloading new software to a system. This method is used by software development tools, such as the Nios II integrated development environment (IDE). There is little point in downloading a program to a target hardware system, if the program is compiled for different hardware. Therefore, the Nios II IDE checks that the system ID core in hardware matches the expected system ID of the software before downloading a program to run or debug.
- Check system ID after reset. If a program is running on hardware other than the expected Platform Designer system, the program may fail to function altogether. If the program does not crash, it can behave erroneously in subtle ways that are difficult to debug. To protect against this case, a program can compare the expected system ID against the system ID core, and report an error if they do not match.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

40.3. Configuration

The `id` and `timestamp` register values are determined at system generation time based on the configuration of the Platform Designer system and the current time. You can add only one system ID core to an Platform Designer system, and its name is always `sysid`.

After system generation, you can examine the values stored in the `id` and `timestamp` registers by opening the IP Parameter Editor for the System ID core.

Since a unique `timestamp` value is added to the System ID HDL file each time you generate the Platform Designer system, the Intel Quartus Prime software recompiles the entire system if you have added the system as a design partition.

Note: In Intel Quartus Prime Pro Edition, the Platform Designer generation process needs an additional TCL script for manual execution to have a unique `timestamp` value.

Follow these steps to have a unique `timestamp` value using the Intel Quartus Prime Pro Edition version:

1. Create a Platform Designer system with the System ID core.
2. Go to **View**>**System Scripting**. Click on **User Scripts** and direct to `soceds/examples/hardware/<*_ghrd folder>/update_sysid.tcl` (you may also copy the script into your local directory and point to it).
3. Click on **Run Script**.
4. After you run the script, the IP Parameter Editor window appears. Ensure that no error message is shown under **System Scripting Messages** box.
5. To save changes to your `.ip` file, select **File** > **Save** option.
6. Select **File** > **Open** to open your main QSYS system. Switch to**System** tab of the Open window, navigate and select your main QSYS system file in the **Platform designer system** field (on the bottom).

   Note: You must change **Files of Type** field to *All Files (*.qsys)*, then click **Open**.
7. Click on **Sync System Info** in Platform Designer.
8. Click on **Generate HDL**.

40.4. Software Programming Model

This section describes the software programming model for the system ID core. For Nios II processor users, Intel provides the HAL system library header file that defines the System ID core registers.

The System ID core comes with the following software files. These files provide low-level access to the hardware. Application developers should not modify these files.

- `alt_avalon_sysid_regs.h`—Defines the interface to the hardware registers.
- `alt_avalon_sysid.c, alt_avalon_sysid.h`—Header and source files defining the hardware access functions.

  Intel provides one access routine, `alt_avalon_sysid_test()`, that returns a value indicating whether the system ID expected by software matches the system ID core.

40.4.1. alt_avalon_sysid_test()

Prototype:	alt_32 alt_avalon_sysid_test(void)
Thread-safe:	No.
Available from ISR:	Yes.
Include:	<altera_avalon_sysid.h>
Description:	Returns 0 if the values stored in the hardware registers match the values expected by software. Returns 1 if the hardware timestamp is greater than the software timestamp. Returns -1 if the software timestamp is greater than the hardware timestamp.

40.5. System ID Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.07.30	18.0	Corrected the steps to have a unique `timestamp` value in *Configuration* section.
2018.05.07	18.0	Added the steps to have a unique `timestamp` value in *Configuration* section.

Date	Version	Changes
July 2014	2014.07.24	Removed mention of SOPC Builder, updated to Platform Designer
December 2010	v10.1.0	Removed the “Device Support”, “Instantiating the Core in SOPC Builder”, and “Referenced Documents” sections.
July 2010	v10.0.0	No change from previous release.
November 2009	v9.1.0	Added description to the Instantiating the Core in SOPC Builder section.
March 2009	v9.0.0	No change from previous release.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	v8.0.0	No change from previous release.

41. Avalon Packets to Transactions Converter Core

41.1. Core Overview

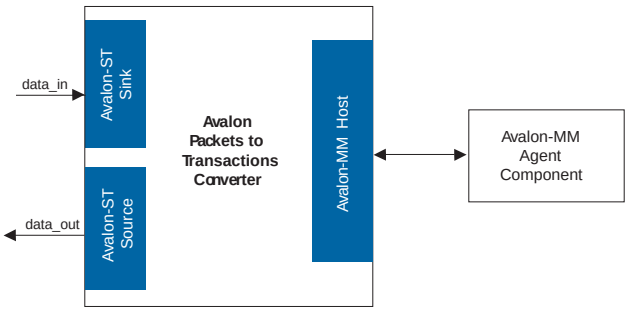
The Avalon Packets to Transactions Converter core receives streaming data from upstream components and initiates Avalon Memory-Mapped (Avalon-MM) transactions. The core then returns Avalon-MM transaction responses to the requesting components.

The SPI Agent to Avalon Host Bridge and JTAG to Avalon Host Bridge are examples of how this core is used.

For more information on the bridge, refer to "SPI Agent/JTAG to Avalon Host Bridge Cores" on page 18–1

41.2. Functional Description

Figure 126. Avalon Packets to Transactions Converter Core



41.2.1. Interfaces

Table 420. Properties of Avalon-ST Interfaces

Feature	Property
Backpressure	Ready latency = 0.
Data Width	Data width = 8 bits; Bits per symbol = 8.
Channel	Not supported.
Error	Not used.
Packet	Supported.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

The Avalon-MM Host interface supports read and write transactions. The data width is set to 32 bits and burst transactions are not supported.

For more information about Avalon-ST interfaces, refer to Avalon Interface Specifications.

41.2.2. Operation

The Avalon Packets to Transactions Converter core receives streams of packets on its Avalon-ST sink interface and initiates Avalon-MM transactions. Upon receiving transaction responses from Avalon-MM agents, the core transforms the responses to packets and returns them to the requesting components via its Avalon-ST source interface. The core does not report Avalon-ST errors.

Packet Formats

The core expects incoming data streams to be in the format shown in the table below. A response packet is returned for every write transaction. The core also returns a response packet if a no transaction (0x7f) is received. An invalid transaction code is regarded as a no transaction. For read transactions, the core simply returns the data read.

Table 421. Packet Formats

Byte	Field	Description
Transaction Packet Format		
0	`Transaction code`	Type of transaction. See **Properties of Avalon-ST Interfaces** table.
1	Reserved	Reserved for future use.
[3:2]	`Size`	Transaction size in bytes. For write transactions, the size indicates the size of the `data` field. For read transactions, the size indicates the total number of bytes to read.
[7:4]	`Address`	32-bit address for the transaction.
[n:8]	`Data`	Transaction data; data to be written for write transactions.
Response Packet Format		
0	`Transaction code`	The transaction code with the most significant bit inversed.
1	Reserved	Reserved for future use.
[3:2]	`Size`	Total number of bytes read/written successfully.

Supported Transactions

The table below lists the Avalon-MM transactions supported by the core.

Table 422. Transaction Supported

Transaction Code	Avalon-MM Transaction	Description
`0x00`	Write, non-incrementing address.	Writes data to the given address until the total number of bytes written to the same word address equals to the value specified in the `size` field.
`0x04`	Write, incrementing address.	Writes transaction data starting at the given address.

continued...

Transaction Code	Avalon-MM Transaction	Description
0x10	Read, non-incrementing address.	Reads 32 bits of data from the given address until the total number of bytes read from the same address equals to the value specified in the size field.
0x14	Read, incrementing address.	Reads the number of bytes specified in the size field starting from the given address.
0x7f	No transaction.	No transaction is initiated. You can use this transaction type for testing purposes. Although no transaction is initiated on the Avalon-MM interface, the core still returns a response packet for this transaction code.

The core can handle only a single transaction at a time. The ready signal on the core's Avalon-ST sink interface is asserted only when the current transaction is completely processed.

No internal buffer is implemented on the data paths. Data received on the Avalon-ST interface is forwarded directly to the Avalon-MM interface and vice-versa. Asserting the waitrequest signal on the Avalon-MM interface back-pressures the Avalon-ST sink interface. In the opposite direction, if the Avalon-ST source interface is back-pressured, the read signal on the Avalon-MM interface is not asserted until the backpressure is alleviated. Back-pressuring the Avalon-ST source in the middle of a read could result in data loss. In such cases, the core returns the data that is successfully received.

A transaction is considered complete when the core receives an EOP. For write transactions, the actual data size is expected to be the same as the value of the size field. Whether or not both values agree, the core always uses the EOP to determine the end of data.

Malformed Packets

The following are examples of malformed packets:

- Consecutive start of packet (SOP)—An SOP marks the beginning of a transaction. If an SOP is received in the middle of a transaction, the core drops the current transaction without returning a response packet for the transaction, and initiates a new transaction. This effectively handles packets without an end of packet(EOP).
- Unsupported transaction codes—The core treats unsupported transactions as a no transaction.

41.3. Avalon Packets to Transactions Converter Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
November 2017	2017.11.06	Corrected the Size field for Response Packet Format in the *Table: Packet Formats*.
July 2014	2014.07.24	Removed mention of SOPC Builder, updated to Qsys.

continued...

Date	Version	Changes
December 2010	v10.1.0	Removed the "Device Support", "Instantiating the Core in SOPC Builder", and "Referenced Documents" sections.
July 2010	v10.0.0	No change from previous release.
November 2009	v9.1.0	No change from previous release.
March 2009	v9.0.0	No change from previous release.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	v8.0.0	Initial release.

42. Avalon-ST Multiplexer and Demultiplexer Cores

42.1. Core Overview

The Avalon streaming (Avalon-ST) channel multiplexer core receives data from a number of input interfaces and multiplexes the data into a single output interface, using the optional `channel` signal to indicate which input the output data is from. The Avalon-ST channel demultiplexer core receives data from a channelized input interface and drives that data to multiple output interfaces, where the output interface is selected by the input `channel` signal.

The multiplexer and demultiplexer can transfer data between interfaces on cores that support the unidirectional flow of data. The multiplexer and demultiplexer allow you to create multiplexed or de-multiplexer datapaths without having to write custom HDL code to perform these functions. The multiplexer includes a round-robin scheduler. Both cores are Platform Designer-ready and integrate easily into any Platform Designer-generated system. This chapter contains the following sections:

42.1.1. Resource Usage and Performance

Resource utilization for the cores depends upon the number of input and output interfaces, the width of the datapath and whether the streaming data uses the optional packet protocol. For the multiplexer, the parameterization of the scheduler also effects resource utilization.

Table 423. Multiplexer Estimated Resource Usage and Performance

No. of Inputs	Data Width	Scheduling Size (Cycles)	Stratix II and Stratix II GX (Approximate LEs)		Cyclone II		Stratix	
			f_{MAX} (MHz)	ALM Count	f_{MAX} (MHz)	Logic Cells	f_{MAX} (MHz)	Logic Cells
2	Y	1	500	31	420	63	422	80
2	Y	2	500	36	417	60	422	58
2	Y	32	451	43	364	68	360	49
8	Y	2	401	150	257	233	228	298
8	Y	32	356	151	219	207	211	123
16	Y	2	262	333	174	533	170	284
16	Y	32	310	337	161	471	157	277
2	N	1	500	23	400	48	422	52

continued...

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

No. of Inputs	Data Width	Scheduling Size (Cycles)	Stratix II and Stratix II GX (Approximate LEs)		Cyclone II		Stratix	
			f_{MAX} (MHz)	ALM Count	f_{MAX} (MHz)	Logic Cells	f_{MAX} (MHz)	Logic Cells
2	N	9	500	30	420	52	422	56
11	N	9	292	275	197	397	182	287
16	N	9	262	295	182	441	179	224

The core operating frequency varies with the device, the number of interfaces and the size of the datapath.

Table 424. Demultiplexer Estimated Resource Usage

No. of Inputs	Data Width (Symbols per Beat)	Stratix II (Approximate LEs)		Cyclone II		Stratix II GX (Approximate LEs)	
		f_{MAX} (MHz)	ALM Count	f_{MAX} (MHz)	Logic Cells	f_{MAX} (MHz)	Logic Cells
2	1	500	53	400	61	399	44
15	1	349	171	235	296	227	273
16	1	363	171	233	294	231	290
2	2	500	85	392	97	381	71
15	2	352	247	213	450	210	417
16	2	328	280	218	451	222	443

42.2. Multiplexer

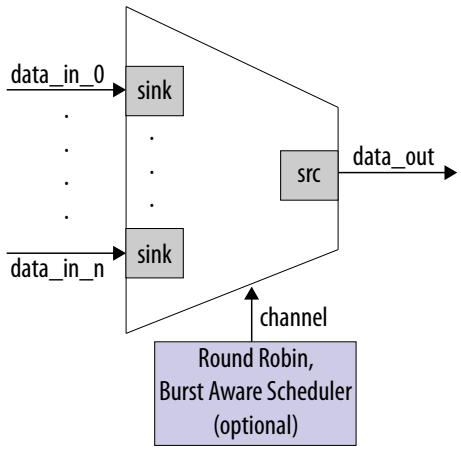
This section describes the hardware structure and functionality of the multiplexer component.

42.2.1. Functional Description

The Avalon-ST multiplexer takes data from a number of input data interfaces, and multiplexes the data onto a single output interface. The multiplexer includes a simple, round-robin scheduler that selects from the next input interface that has data. Each input interface has the same width as the output interface, so that all other input interfaces are backpressured when the multiplexer is carrying data from a different input interface.

The multiplexer includes an optional `channel` signal that enables each input interface to carry channelized data. When the `channel` signal is present on input interfaces, the multiplexer adds $\log_2$ (num_input_interfaces) bits to make the output channel signal, such that the output channel signal has all of the bits of the input channel plus the bits required to indicate which input interface each cycle of data is from. These bits are appended to either the most or least significant bits of the output `channel` signal as specified in the Platform Designer MegaWizard™ interface.

Figure 127. Multiplexer



The internal scheduler considers one input interface at a time, selecting it for transfer. Once an input interface has been selected, data from that input interface is sent until one of the following scenarios occurs:

- The specified number of cycles have elapsed.
- The input interface has no more data to send and `valid` is deasserted on a ready cycle.
- When packets are supported, `endofpacket` is asserted.

Input Interfaces

Each input interface is an Avalon-ST data interface that optionally supports packets. The input interfaces are identical; they have the same symbol and data widths, error widths, and channel widths.

Output Interface

The output interface carries the multiplexed data stream with data from all of the inputs. The symbol, data, and error widths are the same as the input interfaces. The width of the `channel` signal is the same as the input interfaces, with the addition of the bits needed to indicate the input each datum was from.

42.2.2. Parameters

The following sections list the available options in the IP Parameter Editor.

Functional Parameters

You can configure the following options for the multiplexer:

- **Number of Input Ports**—The number of input interfaces that the multiplexer supports. Valid values are 2–16.
- **Scheduling Size (Cycles)**—The number of cycles that are sent from a single channel before changing to the next channel.
- **Use Packet Scheduling**—When this option is on, the multiplexer only switches the selected input interface on packet boundaries. Hence, packets on the output interface are not interleaved.
- **Use high bits to indicate source port**—When this option is on, the high bits of the output channel signal are used to indicate the input interface that the data came from. For example, if the input interfaces have 4-bit channel signals, and the multiplexer has 4 input interfaces, the output interface has a 6-bit channel signal. If this parameter is true, bits [5:4] of the output channel signal indicate the input interface the data is from, and bits [3:0] are the channel bits that were presented at the input interface.

Output Interface

You can configure the following options for the output interface:

- **Data Bits Per Symbol**—The number of bits per symbol for the input and output interfaces. Valid values are 1–32 bits.
- **Data Symbols Per Beat**—The number of symbols (words) that are transferred per beat (transfer). Valid values are 1–32.
- **Include Packet Support**—Indicates whether or not packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
- **Channel Signal Width (bits)**—The number of bits used for the `channel` signal for input interfaces. A value of 0 indicates that input interfaces do not have channels. A value of 4 indicates that up to 16 channels share the same input interface. The input channel can have a width between 0–31 bits. A value of 0 means that the optional `channel` signal is not used.
- **Error Signal Width (bits)**—The width of the `error` signal for input and output interfaces. A value of 0 means the `error` signal is not used.

42.3. Demultiplexer

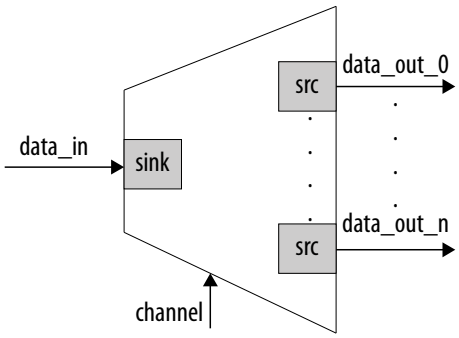
This section describes the hardware structure and functionality of the demultiplexer component.

42.3.1. Functional Description

That Avalon-ST demultiplexer takes data from a channelized input data interface and provides that data to multiple output interfaces, where the output interface selected for a particular transfer is specified by the input `channel` signal. The data is delivered to the output interfaces in the same order it was received at the input interface, regardless of the value of `channel`, `packet`, `frame`, or any other signal. Each of the output interfaces has the same width as the input interface, so each output interface is idle when the demultiplexer is driving data to a different output interface. The

demultiplexer uses $\log_2$ (num_output_interfaces) bits of the `channel` signal to select the output to which to forward the data; the remainder of the channel bits are forwarded to the appropriate output interface unchanged.

Figure 128. Demultiplexer



Input Interface

Each input interface is an Avalon-ST data interface that optionally supports packets.

Output Interfaces

Each output interface carries data from a subset of channels from the input interface. Each output interface is identical; all have the same symbol and data widths, error widths, and channel widths. The symbol, data, and error widths are the same as the input interface. The width of the `channel` signal is the same as the input interface, without the bits that were used to select the output interface.

42.3.2. Parameters

The following sections list the available options in the IP Parameter Editor.

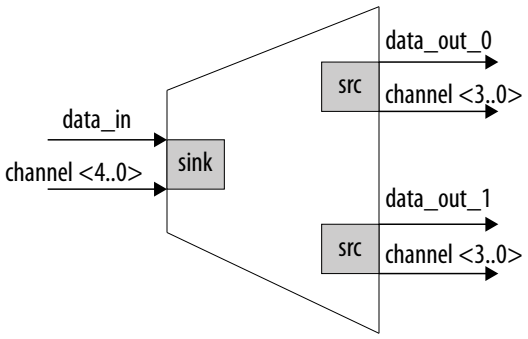
Functional Parameters

You can configure the following options for the demultiplexer as a whole:

- **Number of Output Ports**—The number of output interfaces that the multiplexer supports Valid values are 2–16.
- **High channel bits select output**—When this option is on, the high bits of the input channel signal are used by the de-multiplexing function and the low order bits are passed to the output. When this option is off, the low order bits are used and the high order bits are passed through.

  The following example illustrates the significance of the location of these signals. In the **Select Bits for De-multiplexer** figure below there is one input interface and two output interfaces. If the low-order bits of the channel signal select the output interfaces, the even channels goes to channel 0 and the odd channels goes to channel 1. If the high-order bits of the channel signal select the output interface, channels 0–7 goes to channel 0 and channels 8–15 goes to channel 1.

Figure 129. Select Bits for De-multiplexer



Input Interface

You can configure the following options for the input interface:

- **Data Bits Per Symbol**—The number of bits per symbol for the input and output interfaces. Valid values are 1 to 32 bits.
- **Data Symbols Per Beat**—The number of symbols (words) that are transferred per beat (transfer). Valid values are 1 to 32.
- **Include Packet Support**—Indicates whether or not packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
- **Channel Signal Width (bits)**—The number of bits used for the `channel` signal for output interfaces. A value of 0 means that output interfaces do not use the optional `channel` signal.
- **Error Signal Width (bits)**—The width of the `error` signal for input and output interfaces. A value of 0 means the `error` signal is not unused.

42.4. Hardware Simulation Considerations

The multiplexer and demultiplexer components do not provide a simulation testbench for simulating a stand-alone instance of the component. However, you can use the standard Platform Designer simulation flow to simulate the component design files inside an Platform Designer system.

42.5. Software Programming Model

The multiplexer and demultiplexer components do not have any user-visible control or status registers. Therefore, software cannot control or configure any aspect of the multiplexer or de-multiplexer at run-time. The components cannot generate interrupts.

42.6. Avalon-ST Multiplexer and Demultiplexer Cores Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
December 2010	v10.1.0	Removed the "Device Support", "Instantiating the Core in SOPC Builder", and "Referenced Documents" sections.
July 2010	v10.0.0	No change from previous release.
November 2009	v9.1.0	No change from previous release.
March 2009	v9.0.0	No change from previous release.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. Added parameter **Include Packet Support**.
May 2008	v8.0.0	No change from previous release.

43. Avalon-ST Bytes to Packets and Packets to Bytes Converter Cores

43.1. Core Overview

The Avalon Streaming (Avalon-ST) Bytes to Packets and Packets to Bytes Converter cores allow an arbitrary stream of packets to be carried over a byte interface, by encoding packet-related control signals such as `startofpacket` and `endofpacket` into byte sequences.The Avalon-ST Packets to Bytes Converter core encodes packet control and payload as a stream of bytes. The Avalon-ST Bytes to Packets Converter core accepts an encoded stream of bytes, and converts it into a stream of packets.

The SPI Agent to Avalon host Bridge and JTAG to Avalon Host Bridge are examples of how the cores are used.

For more information about the bridge, refer to SPI Agent/JTAG to Avalon Host Bridge Cores

43.2. Functional Description

The following two figures show block diagrams of the Avalon-ST Bytes to Packets and Packets to Bytes Converter cores.

Figure 130. Avalon-ST Bytes to Packets Converter Core

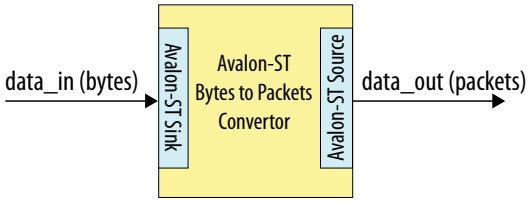
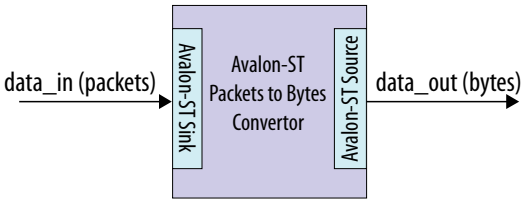


Figure 131. Avalon-ST Packets to Bytes Converter Core



Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

43.2.1. Interfaces

Table 425. Properties of Avalon-ST Interfaces

Feature	Property
Backpressure	Ready latency = 0.
Data Width	Data width = 8 bits; Bits per symbol = 8.
Channel	Supported, up to 255 channels.
Error	Not used.
Packet	Supported only on the Avalon-ST Bytes to Packet Converter core's source interface and the Avalon-ST Packet to Bytes Converter core's sink interface.

For more information about Avalon-ST interfaces, refer to the Avalon Interface Specifications.

43.2.2. Operation—Avalon-ST Bytes to Packets Converter Core

The Avalon-ST Bytes to Packets Converter core receives streams of bytes and transforms them into packets. When parsing incoming bytestreams, the core decodes special characters in the following manner, with higher priority operations listed first:

- Escape (`0x7d`)—The core drops the byte. The next byte is XOR'ed with `0x20`.
- Start of packet (`0x7a`)—The core drops the byte and marks the next payload byte as the start of a packet by asserting the `startofpacket` signal on the Avalon-ST source interface.
- End of packet (`0x7b`)—The core drops the byte and marks the following byte as the end of a packet by asserting the `endofpacket` signal on the Avalon-ST source interface. For single beat packets, both the `startofpacket` and `endofpacket` signals are asserted in the same clock cycle.

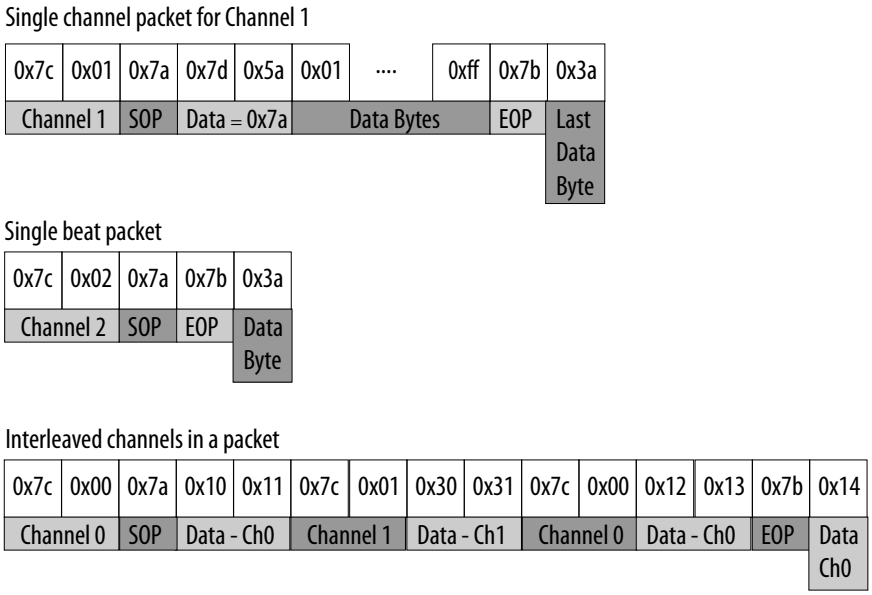
  There are two possible cases if the payload is a special character:

  — The byte sent after end of packet is ESC'ed and XOR'ed with `0x20`.

  — The byte sent after end of packet is assumed to be the last byte regardless of whether or not it is a special character.

  Note: The escape character should be used after an end of packet if the next character requires it.
- Channel number indicator (`0x7c`)—The core drops the byte and takes the next non-special character as the channel number.

Figure 132. Examples of Bytestreams



43.2.3. Operation—Avalon-ST Packets to Bytes Converter Core

The Avalon-ST Packets to Bytes Converter core receives packetized data and transforms the packets to bytestreams. The core constructs outgoing bytestreams by inserting appropriate special characters in the following manner and sequence:

- If the `startofpacket` signal on the core's source interface is asserted, the core inserts the following special characters:
  - Channel number indicator (`0x7c`).
  - Channel number, escaping it if required.
  - Start of packet (`0x7a`).
- If the `endofpacket` signal on the core's source interface is asserted, the core inserts an end of packet (`0x7b`) before the last byte of data.
- If the `channel` signal on the core's source interface changes to a new value within a packet, the core inserts a channel number indicator (`0x7c`) followed by the new channel number.
- If a data byte is a special character, the core inserts an escape (`0x7d`) followed by the data XORed with `0x20`.

43.3. Avalon-ST Bytes to Packets and Packets to Bytes Converter Cores Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
November 2015	2015.11.06	Updated "Operation-Avalon-ST Bytes to Packets Converter Core" section.
July 2014	2014.07.24	Removed mention of SOPC Builder, updated to Platform Designer.
December 2010	v10.1.0	Removed the “Device Support”, “Instantiating the Core in SOPC Builder”, and “Referenced Documents” sections.
July 2010	v10.0.0	No change from previous release.
November 2009	v9.1.0	No change from previous release.
March 2009	v9.0.0	No change from previous release.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	v8.0.0	Initial release.

44. Avalon-ST Delay Core

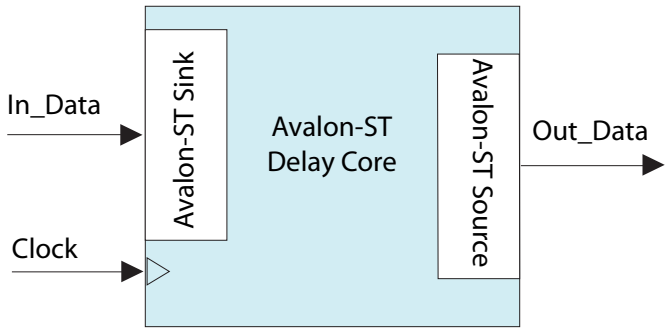
44.1. Core Overview

The Avalon Streaming (Avalon-ST) Delay core provides a solution to delay Avalon-ST transactions by a constant number of clock cycles. This core supports up to 16 clock cycle delays.

The Avalon-ST Delay core is Platform Designer-ready and integrates easily into any Platform Designer-generated system.

44.2. Functional Description

Figure 133. Avalon-ST Delay Core



The Avalon-ST Delay core adds a delay between the input and output interfaces. The core accepts all transactions presented on the input interface and reproduces them on the output interface N cycles later without changing the transaction.

The input interface delays the input signals by a constant (N) number of clock cycles to the corresponding output signals of the Avalon-ST output interface. The **Number Of Delay Clocks** parameter defines the constant (N) number, which must be between 0 and 16. The change of the `In_Valid` signal is reflected on the `Out_Valid` signal exactly N cycles later.

44.2.1. Reset

The Avalon-ST Delay core has a `reset` signal that is synchronous to the `clk` signal. When the core asserts the `reset` signal, the output signals are held at 0. After the `reset` signal is deasserted, the output signals are held at 0 for N clock cycles. The delayed values of the input signals are then reflected at the output signals after N clock cycles.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

44.2.2. Interfaces

The Avalon-ST Delay core supports packetized and non-packetized interfaces with optional channel and error signals. This core does not support backpressure.

Table 426. Properties of Avalon-ST Interfaces

Feature	Property
Backpressure	Not supported.
Data Width	Configurable.
Channel	Supported (optional).
Error	Supported (optional).
Packet	Supported (optional).

For more information about Avalon-ST interfaces, refer to the Avalon Interface Specifications.

44.3. Parameters

Table 427. Configurable Parameters

Parameter	Legal Values	Default Value	Description
Number Of Delay Clocks	0 to 16	1	Specifies the delay the core introduces, in clock cycles. The value of 0 is supported for some cases of parameterized systems in which no delay is required.
Data Width	1–512	8	The width of the data on the Avalon-ST data interfaces.
Bits Per Symbol	1–512	8	The number of bits per symbol for the input and output interfaces. For example, byte-oriented interfaces have 8-bit symbols.
Use Packets	0 or 1	0	Indicates whether or not packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
Use Channel	0 or 1	0	The option to enable or disable the channel signal.
Channel Width	0-8	1	The width of the `channel` signal on the data interfaces. This parameter is disabled when **Use Channel** is set to 0.
Max Channels	0-255	1	The maximum number of channels that a data interface can support. This parameter is disabled when **Use Channel** is set to 0.
Use Error	0 or 1	0	The option to enable or disable the error signal.
Error Width	0–31	1	The width of the `error` signal on the output interfaces. A value of 0 indicates that the error signal is not in use. This parameter is disabled when **Use Error** is set to 0.
Use packets	0 or 1		Setting this parameter to 1 enables packet support on the Avalon-ST data interfaces.
Use fill level	0 or 1		Setting this parameter to 1 enables the Avalon-MM status interface.

continued...

Parameter	Legal Values	Default Value	Description
Number of almost-full thresholds	0 to 2		The number of almost-full thresholds to enable. Setting this parameter to 1 enables **Use almost-full threshold 1**. Setting it to 2 enables both **Use almost-full threshold 1** and **Use almost-full threshold 2**.
Number of almost-empty thresholds	0 to 2		The number of almost-empty thresholds to enable. Setting this parameter to 1 enables **Use almost-empty threshold 1**. Setting it to 2 enables both **Use almost-empty threshold 1** and **Use almost-empty threshold 2**.
Section available threshold	0 to 2 Address Width		Specify the amount of data to be delivered to the output interface. This parameter applies only when packet support is disabled.
Packet buffer mode	0 or 1		Setting this parameter to 1 causes the core to deliver only full packets to the output interface. This parameter applies only when **Use packets** is set to 1.
Drop on error	0 or 1		Setting this parameter to 1 causes the core to drop packets at the Avalon-ST data sink interface if the `error` signal on that interface is asserted. Otherwise, the core accepts the packet and sends it out on the Avalon-ST data source interface with the same error. This parameter applies only when packet buffer mode is enabled.
Use almost-full threshold 1	0 or 1		This threshold indicates that the FIFO is almost full. It is enabled when the parameter **Number of almost-full threshold** is set to 1 or 2.
Use almost-full threshold 2	0 or 1		This threshold is an initial indication that the FIFO is getting full. It is enabled when the parameter **Number of almost-full threshold** is set to 2.
Use almost-empty threshold 1	0 or 1		This threshold indicates that the FIFO is almost empty. It is enabled when the parameter **Number of almost-empty threshold** is set to 1 or 2.
Use almost-empty threshold 2	0 or 1		This threshold is an initial indication that the FIFO is getting empty. It is enabled when the parameter **Number of almost-empty threshold** is set to 2.

44.4. Avalon-ST Delay Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
December 2010	v10.1.0	Removed the “Device Support”, “Instantiating the Core in SOPC Builder”, and “Referenced Documents” sections.
July 2010	v10.0.0	No change from previous release.
January 2010	v9.1.1	Initial release.

45. Avalon-ST Round Robin Scheduler Core

45.1. Core Overview

Avalon Streaming (Avalon-ST) components in Platform Designer provide a channel interface to stream data from multiple channels into a single component. In a multi-channel Avalon-ST component that stores data, the component can store data either in the sequence that it comes in (FIFO) or in segments according to the channel. When data is stored in segments according to channels, a scheduler is needed to schedule the read operations from that particular component. The most basic of the schedulers is the Avalon-ST Round Robin Scheduler core.

The Avalon-ST Round Robin Scheduler core is Platform Designer-ready and can integrate easily into any Platform Designer-generated systems.

45.2. Performance and Resource Utilization

This section lists the resource utilization and performance data for various Intel FPGA device families. The estimates are obtained by compiling the core using the Intel Quartus Prime software.

The table below shows the resource utilization and performance data for a Stratix II GX device (EP2SGX130GF1508I4).

Table 428. Resource Utilization and Performance Data for Stratix II GX Devices

Number of Channels	ALUTs	Logic Registers	Memory M512/M4K/ M-RAM	f_{MAX} (MHz)
4	7	7	0/0/0	> 125
12	25	17	0/0/0	> 125
24	62	30	0/0/0	> 125

The table below shows the resource utilization and performance data for a Stratix III device (EP3SL340F1760C3). The performance of the IP Core in Stratix IV devices is similar to Stratix III devices.

Table 429. Resource Utilization and Performance Data for Stratix III Devices

Number of Channels	ALUTs	Logic Registers	Memory M9K/ M144K/ MLAB	f_{MAX} (MHz)
4	7	7	0/0/0	> 125
12	25	17	0/0/0	> 125
24	67	30	0/0/0	> 125

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

The table below shows the resource utilization and performance data for a Cyclone III device (EP3C120F780I7).

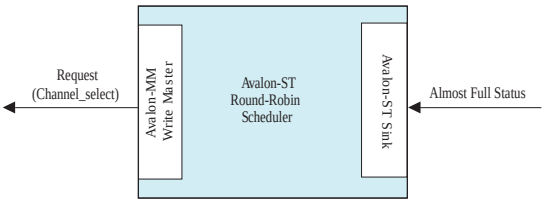
Table 430. Resource Utilization and Performance Data for Cyclone III Devices

Number of Channels	Total Logic Elements	Total Registers	Memory M9K	f_{MAX} (MHz)
4	12	7	0	> 125
12	32	17	0	> 125
24	71	30	0	> 125

45.3. Functional Description

The Avalon-ST Round Robin Scheduler core controls the read operations from a multi-channel Avalon-ST component that buffers data by channels. It reads the almost-full threshold values from the multiple channels in the multi-channel component and issues the read request to the Avalon-ST source according to a round-robin scheduling algorithm.

Figure 134. Avalon-ST Round Robin Scheduler Block Diagram



45.3.1. Interfaces

The following interfaces are available in the Avalon-ST Round Robin Scheduler core:

- Almost-Full Status Interface
- Request Interface

Almost-Full Status Interface

The Almost-Full Status interface is an Avalon-ST sink interface.

Table 431. Avalon-ST Interface Feature Support

Feature	Property
Backpressure	Not supported
Data Width	Data width = 1; Bits per symbol = 1

continued...

Feature	Property
Channel	Maximum channel = 32; Channel width = 5
Error	Not supported
Packet	Not supported

The interface collects the almost-full status from the sink components for all the channels in the sequence provided.

Request Interface

The Request Interface is an Avalon Memory-Mapped (MM) Write Host interface. This interface requests data from a specific channel. The Avalon-ST Round Robin Scheduler core cycles through all of the channels it supports and schedules data to be read.

45.3.2. Operations

If a particular channel is almost full, the Avalon-ST Round Robin Scheduler will not schedule data to be read from that channel in the source component.

The Avalon-ST Round Robin Scheduler only requests 1 beat of data from a channel at each transaction. To request 1 beat of data from channel n, the scheduler writes the value 1 to address (4 ×n). For example, if the scheduler is requesting data from channel 3, the scheduler writes 1 to address 0xC.

At every clock cycle, the Avalon-ST Round Robin Scheduler requests data from the next channel. Therefore, if the Avalon-ST Round Robin Scheduler starts requesting from channel 1, at the next clock cycle, it requests from channel 2. The Avalon-ST Round Robin Scheduler does not request data from a particular channel if the almost-full status for the channel is asserted. In this case, one clock cycle is used without a request transaction.

The Avalon-ST Round Robin Scheduler cannot determine if the requested component is able to service the request transaction. The component asserts `waitrequest` when it cannot accept new requests.

Table 432. Ports for the Avalon-ST Round Robin Scheduler

Signal	Direction	Description
Clock and Reset		
`clk`	In	Clock reference.
`reset_n`	In	Asynchronous active low reset.
Avalon-MM Request Interface		
`request_address` ($\log_2$ `Max_Channels-1:0`)	Out	The write address used to signal the channel the request is for.
`request_write`	Out	Write enable signal.
`request_writedata`	Out	The amount of data requested from the particular channel. This value is always fixed at 1.
`request_waitrequest`	In	Wait request signal, used to pause the scheduler when the agent cannot accept a new request.

continued...

Signal	Direction	Description
Avalon-ST Almost-Full Status Interface		
`almost_full_valid`	In	Indicates that `almost_full_channel` and `almost_full_data` are valid.
`almost_full_channel (Channel_Width–1:0)`	In	Indicates the channel for the current status indication.
`almost_full_data (log2 Max_Channels–1:0)`	In	A 1-bit signal that is asserted high to indicate that the channel indicated by `almost_full_channel` is almost full.

45.4. Parameters

Table 433. Parameters for Avalon-ST Round Robin Scheduler Component

Parameters	Values	Description
Number of channels	2–32	Specifies the number of channels the Avalon-ST Round Robin Scheduler supports.
Use almost-full status	0–1	Specifies whether the almost-full interface is used. If the interface is not used, the core always requests data from the next channel at the next clock cycle.

45.5. Avalon-ST Round Robin Scheduler Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
December 2010	v10.1.0	Removed the “Device Support”, “Instantiating the Core in SOPC Builder”, and “Referenced Documents” sections.
July 2010	v10.0.0	No change from previous release.
November 2009	v9.1.0	No change from previous release.
March 2009	v9.0.0	No change from previous release.
November 2008	v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.
May 2008	v8.0.0	Initial release.

46. Avalon-ST Splitter Core

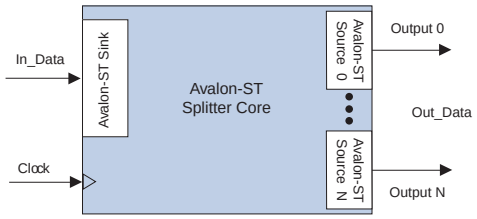
46.1. Core Overview

The Avalon Streaming (Avalon-ST) Splitter core allows you to replicate transactions from an Avalon-ST source interface to multiple Avalon-ST sink interfaces. This core can support from 1 to 16 outputs.

The Avalon-ST Splitter core is Platform Designer-ready and integrates easily into any Platform Designer-generated system.

46.2. Functional Description

Figure 135. Avalon-ST Splitter Core



The Avalon-ST Splitter core copies all input signals from the input interface to the corresponding output signals of each output interface without altering the size or functionality. This include all signals except for the `ready` signal.

The Avalon-ST Splitter core includes a clock signal used by Platform Designer to determine the Avalon-ST interface and clock domain that this core resides in. Because the clock signal is unused internally, no latency is introduced when using this core.

46.2.1. Backpressure

The Avalon Streaming Interface Splitter core handles backpressure by AND-ing the `ready` signals from all of the output interfaces and sending the result to the input interface. This way, if any output interface deasserts the `ready` signal, the input interface receives the deasserted `ready` signal as well. This mechanism ensures that backpressure on any of the output interfaces is propagated to the input interface.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

When the **Qualify Valid Out** parameter is set to 1, the `Out_Valid` signals on all other output interfaces are gated when backpressure is applied from one output interface. In this case, when any output interface deasserts its `ready` signal, the `Out_Valid` signals on the rest of the output interfaces are deasserted as well.

When the **Qualify Valid Out** parameter is set to 0, the output interfaces have a non-gated `Out_Valid` signal when backpressure is applied. In this case, when an output interface deasserts its ready signal, the `Out_Valid` signals on the rest of the output interfaces are not affected.

Because the logic is purely combinational, the core introduces no latency.

46.2.2. Interfaces

The Avalon-ST Splitter core supports packetized and non-packetized interfaces with optional channel and error signals. The core propagates backpressure from any output interface to the input interface.

Table 434. Properties of Avalon-ST Interfaces

Feature	Property
Backpressure	Ready latency = 0.
Data Width	Configurable.
Channel	Supported (optional).
Error	Supported (optional).
Packet	Supported (optional).

For more information about Avalon-ST interfaces, refer to the Avalon Interface Specifications.

46.3. Parameters

Table 435. Configurable Parameters

Parameter	Legal Values	Default Value	Description
Number Of Outputs	1 to 16	2	The number of output interfaces. The value of 1 is supported for some cases of parameterized systems in which no duplicated output is required.
Qualify Valid Out	0 or 1	1	Determines whether the `Out_Valid` signal is gated or non-gated when backpressure is applied.
Data Width	1–512	8	The width of the data on the Avalon-ST data interfaces.
Bits Per Symbol	1–512	8	The number of bits per symbol for the input and output interfaces. For example, byte-oriented interfaces have 8-bit symbols.
Use Packets	0 or 1	0	Indicates whether or not packet transfers are supported. Packet support includes the `startofpacket`, `endofpacket`, and `empty` signals.
Use Channel	0 or 1	0	The option to enable or disable the channel signal.

continued...

Parameter	Legal Values	Default Value	Description
Channel Width	0-8	1	The width of the `channel` signal on the data interfaces. This parameter is disabled when **Use Channel** is set to 0.
Max Channels	0-255	1	The maximum number of channels that a data interface can support. This parameter is disabled when **Use Channel** is set to 0.
Use Error	0 or 1	0	The option to enable or disable the error signal.
Error Width	0–31	1	The width of the `error` signal on the output interfaces. A value of 0 indicates that the error signal is not used. This parameter is disabled when **Use Error** is set to 0.
Use packets	0 or 1		Setting this parameter to 1 enables packet support on the Avalon-ST data interfaces.
Use fill level	0 or 1		Setting this parameter to 1 enables the Avalon-MM status interface.
Number of almost-full thresholds	0 to 2		The number of almost-full thresholds to enable. Setting this parameter to 1 enables **Use almost-full threshold 1**. Setting it to 2 enables both **Use almost-full threshold 1** and **Use almost-full threshold 2**.
Number of almost-empty thresholds	0 to 2		The number of almost-empty thresholds to enable. Setting this parameter to 1 enables **Use almost-empty threshold 1**. Setting it to 2 enables both **Use almost-empty threshold 1** and **Use almost-empty threshold 2**.
Section available threshold	0 to 2 Address Width		Specify the amount of data to be delivered to the output interface. This parameter applies only when packet support is disabled.
Packet buffer mode	0 or 1		Setting this parameter to 1 causes the core to deliver only full packets to the output interface. This parameter applies only when **Use packets** is set to 1.
Drop on error	0 or 1		Setting this parameter to 1 causes the core to drop packets at the Avalon-ST data sink interface if the `error` signal on that interface is asserted. Otherwise, the core accepts the packet and sends it out on the Avalon-ST data source interface with the same error. This parameter applies only when packet buffer mode is enabled.
Use almost-full threshold 1	0 or 1		This threshold indicates that the FIFO is almost full. It is enabled when the parameter **Number of almost-full threshold** is set to 1 or 2.
Use almost-full threshold 2	0 or 1		This threshold is an initial indication that the FIFO is getting full. It is enabled when the parameter **Number of almost-full threshold** is set to 2.
Use almost-empty threshold 1	0 or 1		This threshold indicates that the FIFO is almost empty. It is enabled when the parameter **Number of almost-empty threshold** is set to 1 or 2.
Use almost-empty threshold 2	0 or 1		This threshold is an initial indication that the FIFO is getting empty. It is enabled when the parameter **Number of almost-empty threshold** is set to 2.

46.4. Avalon-ST Splitter Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
December 2010	v10.1.0	Removed the "Device Support", "Instantiating the Core in SOPC Builder", and "Referenced Documents" sections.
July 2010	v10.0.0	No change from previous release.
January 2010	v9.1.1	Initial release.

47. Avalon-MM DDR Memory Half Rate Bridge Core

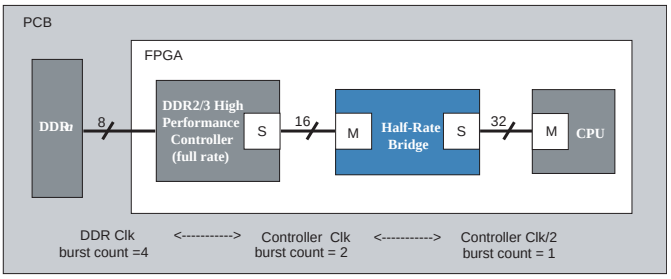
Attention: The Avalon-MM DDR Memory Half Rate Bridge Core is part of a product obsolescence and support discontinuation schedule. For new design, Intel recommends that you use other IPs with equivalent functions. To see a list of available IPs, refer to the Intel FPGA IP Portfolio page.

47.1. Core Overview

The Avalon Memory-Mapped (MM) Half-Rate Bridge core is a special-purpose clock-crossing bridge intended for CPUs that require low-latency access to high-speed memory. The core works under the assumption that the memory clock is twice the frequency of the CPU clock, with zero phase shift between the two. It allows high speed memory to run at full rate while providing low-latency interface for a CPU to access it by using lightweight logic that translates one single-word request into a two-word burst to a memory running at twice the clock frequency and half the width. For systems with a 8-bit DDR interface, using the Half-Rate DDR Bridge in conjunction with a DDR SDRAM high-performance memory controller creates a datapath that matches the throughput of the DDR memory to the CPU. This half-rate bridge provides the same functionality as the clock crossing bridge, but with significantly lower latency —2 cycles instead of 12.

The core's host interface is designed to be connected to a high-speed DDR SDRAM controller and thus only supports bursting. Because the agent interface is designed to receive single-word requests, it does not support bursting. The figure below shows a system including an 8-bit DDR memory, a high-performance memory controller, the Half-Rate DDR Bridge, and a CPU.

Figure 136. Platform Designer Memory System Using a DDR Memory Half-Rate Bridge



Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

The Avalon-MM DDR Memory Half-Rate Bridge core has the following features and requirements:

- Platform Designer ready with Timing Analyzer Timing Analyzer constraints
- Requires host clock and agent clock to be synchronous
- Handles different bus sizes between CPU and memory
- Requires the frequency of the host clock to be double of the agent clock
- Has configurable address and data port widths in the host interface

47.2. Resource Usage and Performance

This section lists the resource usage and performance data for supported devices when operating the Half-Rate Bridge with a full-rate DDR SDRAM high-performance memory controller.

Using the Half-Rate Bridge with a full-rate DDR SDRAM high-performance memory controller results an average of 48% performance improvement over a system using a half-rate DDR SDRAM high-performance memory controller in a series of embedded applications. The performance improvement is 62.2% based on the Dhrystone benchmark, and 87.7% when accessing memory bypassing the cache. For memory systems that use the Half-Rate bridge in conjunction with DDR2/3 High Performance Controller, the data throughput is the same on the Half-Rate Bridge host and agent interfaces. The decrease in memory latency on the Half-Rate Bridge agent interface results in higher performance for the processor.

The table below shows the resource usage for Stratix II and Stratix III devices in the Intel Quartus Prime software with a data width of 16 bits, an address span of 24 bits.

Table 436. Resource Utilization Data for Stratix II and Stratix III Devices

Device Family	Combinational ALUTs	ALMs	Logic Register	Embedded Memory
Stratix II	61	134	153	0
Stratix III	60	138	153	0

Table 437. Resource Utilization Data for Cyclone III Devices

Logic Cells (LC)	Logic Register	LUT-only LC	Register-only LC	LUT/Register LCs	Embedded Memory
233	152	33	84	121	0

47.3. Functional Description

The Avalon MM DDR Memory Half Rate Bridge works under two constraints:

- Its memory-side host has a clock frequency that is synchronous (zero phase shift) to, and twice the frequency of, the CPU-side agent.
- Its memory-side host is half as wide as its CPU-side agent.

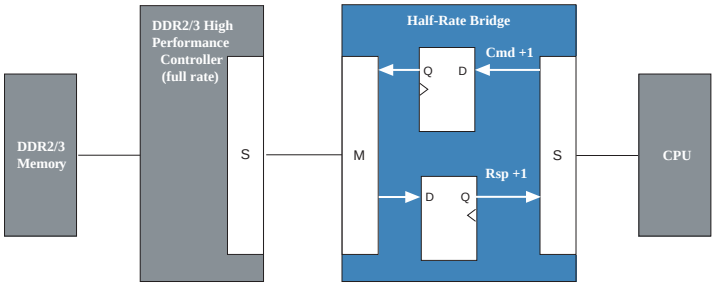
The bridge leverages these two constraints to provide lightweight, low-latency clock-crossing logic between the CPU and the memory. These constraints are in contrast with the Avalon-MM Clock-Crossing Bridge, which makes no assumptions about the

frequency/phase relationship between the host- and agent-side clocks, and provides higher-latency logic that fully-synchronizes all signals that pass between the two domains.

The Avalon MM DDR Memory Half-Rate Bridge has an Avalon-MM agent interface that accepts single-word (non-bursting) transactions. When the agent interface receives a transaction from a connected CPU, it issues a two-word burst transaction on its host interface (which is half as wide and twice as fast). If the transaction is a read request, the bridge's host interface waits for the agent's two-word response, concatenates the two words, and presents them as a single readdata word on its agent interface to the CPU. Every time the data width is halved, the clock rate is doubled. As a result, the data throughput is matched between the CPU and the off-chip memory device.

The figure below shows the latency in the Avalon-MM Half-Rate Bridge core. The core adds two cycles of latency in the agent clock domain for read transactions. The first cycle is introduced during the command phase of the transaction and the second cycle, during the response phase of the transaction. The total latency is 2+<x>, where <x> refers to the latency of the DDR SDRAM high-performance memory controller. Using the clock crossing bridge for this same purpose would impose approximately 12 cycles of additional latency.

Figure 137. Avalon-MM DDR Memory Half-Rate Bridge Block Diagram



47.4. Instantiating the Core in Platform Designer

Use the IP Catalog in Platform Designer to find the Avalon-MM DDR Memory Half-Rate Bridge core. In the parameter editor window you can specify the core's configuration. The table below describes the parameters that can be configured for the Avalon-MM Half-Rate Bridge core.

Table 438. Configurable Parameters for Avalon-MM DDR Memory Half-Rate Bridge Core

Parameters	Allowed Values	Default Value	Description
Data Width	8, 16, 32, 64, 128, 256, 512	16	The width of the data signal in the host interface.
Address Width	1-32	24	The width of the address signal in the host interface.

The table below describes the parameters that are derived based on the **Data Width** and **Address Width** settings for the Avalon-MM DDR Memory Half-Rate Bridge core.

Table 439. Derived Parameters for Avalon-MM DDR Memory Half-Rate Bridge Core

Parameter	Default Value	Description
Host interface's Byte Enable Width	2	The width of the byte-enable signal in the host interface.
Agent interface's Data Width	32	The width of the data signal in the agent interface.
Agent interface's Address Width	22	The width of the address signal in the agent interface.
Agent interface's Byte Enable Width	4	The width of the byte-enable signal in the agent interface.

47.5. Example System

The following example provides high-level steps showing how the Avalon-MM DDR Memory Half-Rate Bridge core is connected in a system. This example assumes that you are familiar with the Platform Designer GUI.

1. Add a **Nios II Processor** to the system.
2. Add a **DDR2 SDRAM High-Performance Controller** and configure it to **full-rate** mode.
3. Add **Avalon-MM DDR Memory Half-Rate Bridge** to the system.
4. Configure the parameters of the Avalon-MM DDR Memory Half-Rate Bridge based on the memory controller. For example, for a 32 MByte DDR memory controller in full rate mode with 8 DQ pins (see Figure 136 on page 546), the parameters should be set as the following:
   - **Data Width** = 16

     For a memory controller that has 8 DQ pins, its local interface width is 16 bits. The local interface width and the data width must be the same, therefore data width is set to 16 bits.
   - **Address Width** = 25

   For a memory capacity of 32 MBytes, the byte address is 25 bits. Because the host address of the bridge is byte aligned, the address width is set to 25 bits.
5. Connect `altmemddr_auxhalf` to the agent clock interface (`clk_s1`) of the Half-Rate Bridge.
6. Connect `altmemddr_sysclk` to the host clock interface (`clk_m1`) of the Half-Rate Bridge.
7. Remove all connections between Nios II processor and the memory controller, if there are any.
8. Connect the host interface (`m1`) of the Avalon-MM DDR Memory Half-Rate Bridge to the memory controller agent interface.
9. Connect the agent interface (`s1`) of the Avalon-MM DDR Memory Half-Rate Bridge to the Nios II processor `data_master` interface.
10. Connect `altmemddr_auxhalf` to Nios II processor clock interface.

47.6. Avalon-MM DDR Memory Half Rate Bridge Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2021.03.29	21.1	Added product obsolescence and support discontinuation notice.
2018.05.07	18.0	Implemented editorial enhancements.
2016.06.17	16.0	Initial release.

48. Intel FPGA GMII to RGMII Converter Core

Note: This IP core only supports Cyclone V SoC devices.

48.1. Core Overview

The Intel FPGA GMII to RGMII converter core is an available soft IP for the FPGA fabric. It converts the GMII/MII interface of the Ethernet controller in the hard processor system (HPS) to an RGMII interface. By default, the HPS Ethernet controller can either provide an RGMII interface on the HPS pins or an GMII/MII interface by using the FPGA loaner I/O. The GMII to RGMII converter also serves as a solution for designers who want to interface to an external RGMII PHY through the FPGA logic..

48.2. Feature Description

48.2.1. Supported Features

The following is the list of features supported by the core.

- Perform GMII/MII interface to RGMII interface conversion
- Supports tri-speed (10/100/1000 Mbps) operation
- Supports dynamic speed switching
- Supports generation time option to enable pipeline registers for the transmit and receive paths

48.2.2. Unsupported Features

The Intel FPGA GMII to RGMII converter core does not support an internal delay of the TX/RX clock. However, the FPGA may still provide the 2 ns delay for center-aligned data transmission/reception through the FPGA I/O buffer. This delay feature is commonly supported by the PHY device or handled at the board level.

For more information on Intel Quartus Prime delay settings, refer to your device's Golden Hardware Reference Design (GHRD) user manual on RocketBoards.org.

Related Information

GSRD User Manual

48.3. Parameters

48.3.1. IP Configuration Parameter

These parameters are configurable by user during generation time.

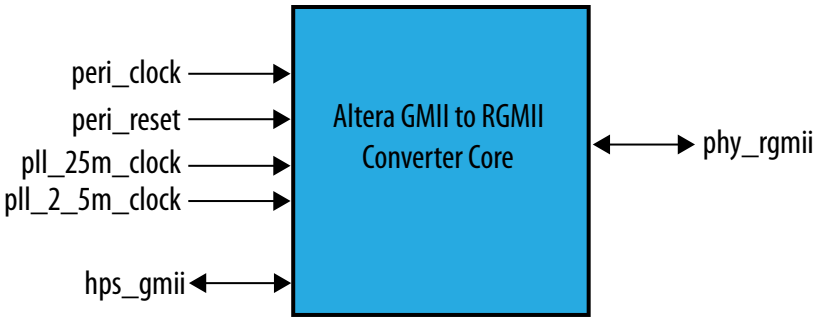
Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Parameter	Legal Values	Default Values	Description
Transmit Pipeline Register Depth	0 - 10	0	**TX_PIPELINE_DEPTH** - Number of register stages between HPS transmit output and FPGA I/O buffer.
Receive Pipeline Register Depth	0 - 10	0	**RX_PIPELINE_DEPTH** - Number of register stages between FPGA I/O buffer and HPS receive input.

48.4. Intel FPGA GMII to RGMII Converter Core Interface

Figure 138. Intel FPGA GMII to RGMII Converter Core Top Level Interfaces



Note: For more information and a detailed list of the interfaces denoted on this figure, refer to the corresponding interface name in the following tables.

Table 440. peri_clock

Interface Name: peri_clock Description: Peripheral clock interface.			
Signal	**Width**	**Direction**	**Description**
clk	1	Input	Peripheral clock source.

Table 441. peri_reset

Interface Name: peri_reset Description: Peripheral reset interface.			
Signal	**Width**	**Direction**	**Description**
rst_n	1	Input	Active low peripheral asynchronous reset source. This signal is asynchronously asserted and synchronously de-asserted. The synchronous de-assertion must be provided external to this core.

Table 442. pll_25m_clock

Interface Name: pll_25m_clock Description: 25MHz clock from FPGA PLL output.			
Signal	**Width**	**Direction**	**Description**
pll_25m_clk	1	Input	25MHz input clock from FPGA PLL.

Table 443. pll_2_5m_clock

Interface Name: pll_2_5m_clock Description: 2.5MHz clock from FPGA PLL output.			
Signal	**Width**	**Direction**	**Description**
pll_2_5m_clk	1	Input	2.5MHz input clock from FPGA PLL.

Table 444. hps_gmii

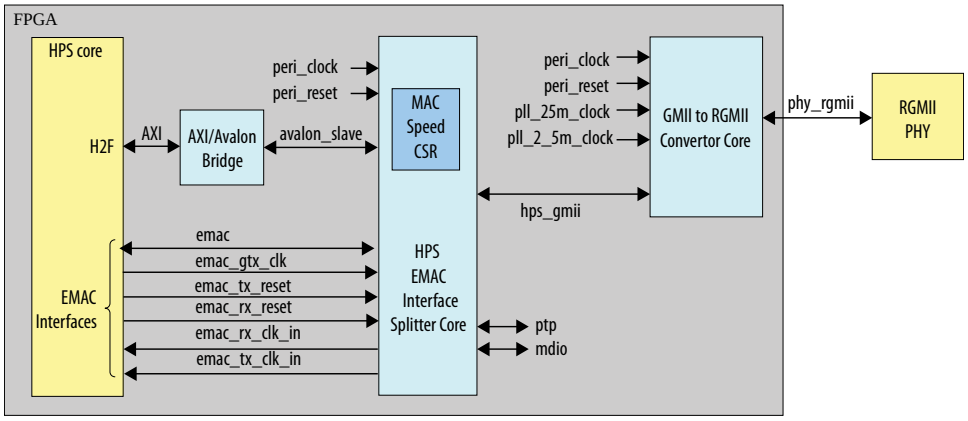
Interface Name: hps_gmii Description: GMII/MII interface facing Intel FPGA HPS Emac Interface Splitter Core			
Signal	**Width**	**Direction**	**Description**
mac_tx_clk_o	1	Input	GMII/MII transmit clock from HPS
mac_tx_clk_i	1	Output	GMII/MII transmit clock to HPS
mac_rx_clk	1	Output	GMII/MII receive clock to HPS
mac_rst_tx_n	1	Input	GMII/MII transmit reset source from HPS. Active low reset
mac_rst_rx_n	1	Input	GMII/MII receive reset source from HPS. Active low reset
mac_txd	8	Input	GMII/MII transmit data from HPS
mac_txen	1	Input	GMII/MII transmit enable from HPS
mac_txer	1	Input	GMII/MII transmit error from HPS
mac_rxdv	1	Output	GMII/MII receive data valid to HPS
mac_rxer	1	Output	GMII/MII receive data error to HPS
mac_rxd	8	Output	GMII/MII receive data to HPS
mac_col	1	Output	GMII/MII collision detect to HPS
mac_crs	1	Output	GMII/MII carrier sense to HPS
mac_speed	2	Input	MAC speed indication from HPS

Table 445. phy_rgmii

Interface Name: phy_rgmii Description: RGMII interface facing PHY device.			
Signal	**Width**	**Direction**	**Description**
rgmii_tx_clk	1	Output	RGMII transmit clock to PHY
rgmii_rx_clk	1	In	RGMII receive clock from PHY
rgmii_txd	4	Output	RGMII transmit data to PHY
rgmii_tx_ctl	1	Output	RGMII transmit control to PHY
rgmii_rxd	4	Input	RGMII receive data from PHY
rgmii_rx_ctl	1	Input	RGMII receive control from PHY

48.5. Functional Description

Figure 139. System Level Block Diagram



Intel FPGA GMII to RGMII converter core is not directly connected to the HPS Ethernet controller. Instead, an intermediate component called the Intel FPGA HPS EMAC interface splitter core is used as a bridge between HPS core and Intel FPGA GMII to RGMII converter core. This intermediate component is responsible for splitting the emac conduit interface output from HPS core into several interfaces according to their function (`hps_gmii`, ptp, mdio interfaces). It is also responsible for managing differences between the EMAC interfaces in the Arria V, Cyclone V, and Intel Arria 10 HPS.

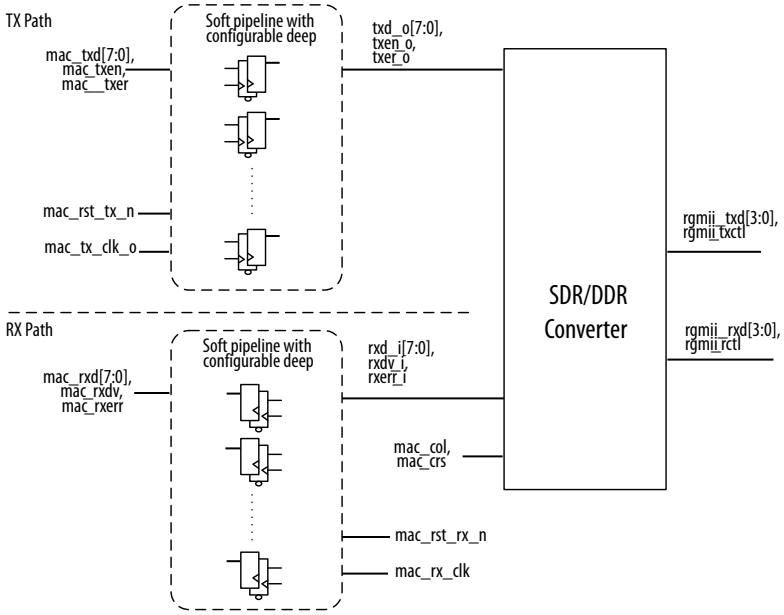
Related Information

Intel FPGA HPS EMAC Interface Splitter Core on page 556
For more information about Intel FPGA HPS EMAC Interface Splitter Core.

48.5.1. Architecture

48.5.1.1. Data Path

Figure 140. Data Path Diagram



For transmit path, the GMII/MII data goes through the transmit pipeline register stage before going into the SDR/DDR converter block. The pipeline logic can be optionally enabled or disabled by the user during generation time.

For receive path, the GMII/MII data right after the SDR/DDR converter block goes directly to EMAC controller through Intel FPGA HPS EMAC interface splitter core; and also goes through the receive pipeline register stage. Similarly, this pipeline logic can be optionally enabled or disabled by the user during generation time.

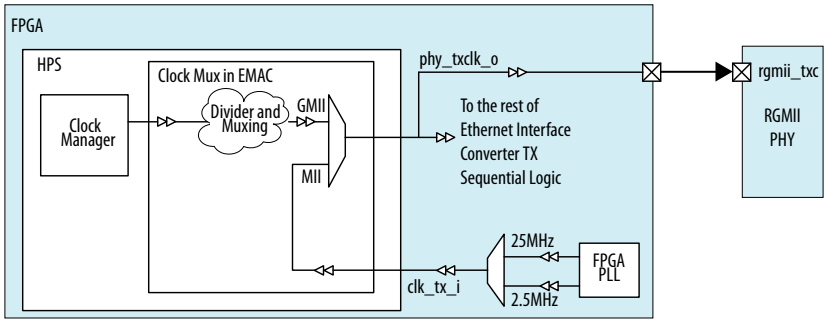
The SDR/DDR converter block manages single data rate to double data rate conversion and vice-versa. Intel FPGA DDIO component (ALTDDIO_IN and ALTDDIO_OUT) is used to perform this task. This block also decodes collision and carrier sense condition through In-Band status detection.

48.5.1.2. Clock Scheme

48.5.1.2.1. Transmit

All transmit sequential logic in the Intel FPGA GMII to RGMII Converter core is clocked by the HPS PLL during GMII mode (1000 Mbps) and by the FPGA PLL during MII mode (10/100 Mbps).

Figure 141. Transmit Clocking Scheme



48.5.1.2.2. Receive

All receive sequential logic in the Intel FPGA GMII to RGMII converter core is clocked by `rgmii_rx_clk` (always driven from the PHY device).

48.6. Intel FPGA HPS EMAC Interface Splitter Core

The Intel FPGA HPS EMAC interface splitter core is used as a bridge between the HPS core and the Intel FPGA GMII to RGMII converter core. It is responsible for splitting the EMAC conduit interface output from the HPS core into several interfaces according to their function (`hps_gmii`, ptp, mdio interfaces). It is also responsible for managing the differences between the EMAC interfaces in the Arria V, Cyclone V, and Intel Arria 10 HPS. Besides the Avalon-MM agent interface logic, there is no additional real logic in this core, except it takes the input signals from HPS, regroups them according to their function, and outputs them.

Note: This IP core supports Intel eASIC N5X devices.

Related Information

Feature Description on page 568

48.6.1. Parameter

48.6.1.1. System Info Parameter

The following parameter is not configurable by the user:

Parameter	Description
DEVICE_FAMILY	**Name:** DEVICE_FAMILY Indicates the device family type of the current selected device in Platform Designer. This parameter is used to determine the version of HPS (Arria V, Cyclone V, and Intel Arria 10 HPS) supported by the current selected device. This information is used to enable or disable certain logic; or to terminate certain interfaces of this core.

48.6.1.2. HDL Parameter

This parameter is not configurable by the user through Platform Designer. Its value is automatically derived by the component based on the DEVICE_FAMILY parameter.

Parameter	Description
Enable mac speed CSR	**Name:** MAC_SPEED_CSR_ENABLE 0: The MAC Speed CSR block is not instantiated in this core. In this case, the Mac Speed information is directly coming from the HPS EMAC interface. 1: The MAC Speed CSR block is instantiated in this core. In this case, the Mac Speed information is determined by the control register defined in this core.

48.6.1.3. Interface Signals

Figure 142. Interface Signals

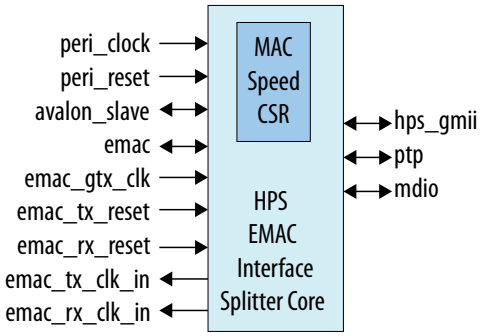


Table 446. peri_clock

Interface Name: peri_clock Description: Peripheral clock interface. This interface exists only when the selected device is Arria V or Cyclone V.			
Signal	**Width**	**Direction**	**Description**
clk	1	Input	Peripheral clock source used for Avalon-MM agent interface.

Table 447. peri_reset

Interface Name: peri_reset Description: Peripheral reset interface. This interface exists only when the selected device is Arria V or Cyclone V.			
Signal	**Width**	**Direction**	**Description**
rst_n	1	Input	Active low peripheral asynchronous reset source used to reset the Avalon-MM agent interface. This signal is asynchronously asserted and synchronously de-asserted. The synchronous de-assertion must be provided external to this core.

Table 448. avalon_slave

Interface Name: avalon_slave Description: This interface exists only when the selected device is Arria V or Cyclone V.			
Signal	**Width**	**Direction**	**Description**
addr	1	Input	Avalon-MM address bus. (40)
read	1	Input	Avalon-MM read control
write	1	Input	Avalon-MM write control
writedata	32	Input	Avalon-MM write data bus
readdata	32	Output	Avalon-MM read data bus

Table 449. emac

Interface Name: emac Description: Conduit interface connected to HPS EMAC interface			
Signal	**Width**	**Direction**	**Description**
phy_txd_o	8	Input	GMII/MII transmit data from HPS
phy_txen_o	1	Input	GMII/MII transmit enable from HPS
phy_txer_o	1	Input	GMII/MII transmit error from HPS
phy_rxdv_i	1	Output	GMII/MII receive data valid to HPS
phy_rxer_i	1	Output	GMII/MII receive data error to HPS
phy_rxd_i	8	Output	GMII/MII receive data to HPS
phy_col_i	1	Output	GMII/MII collision detect to HPS
phy_crs_i	1	Output	GMII/MII carrier sense to HPS
phy_mac_speed_o	2	Input	MAC speed indication from HPS (41)
mdo_o	1	Input	MDIO data output from HPS
mdo_o_e	1	Input	MDIO data output enable from HPS
mdi_i	1	Output	MDIO data input to HPS
ptp_pps_o	1	Input	PTP pulse per second from HPS
ptp_aux_ts_trig_i	1	Output	PTP auxiliary timestamp trigger to HPS

(40) The address bus is in the unit of Word addressing.

(41) These bits exist only when the selected device is Intel Arria 10.

Table 450. emac_gtx_clk

Interface Name: emac_gtx_clk Description: GMII/MII transmit clock from HPS			
Signal	**Width**	**Direction**	**Description**
phy_txclk_o	1	Input	GMII/MII transmit clock from HPS

Table 451. emac_tx_reset

Interface Name: emac_tx_reset Description: GMII/MII transmit reset source synchronous to phy_txclk_o from HPS			
Signal	**Width**	**Direction**	**Description**
rst_tx_n_o	1	Input	GMII/MII transmit reset source from HPS. Active low reset.

Table 452. emac_rx_reset

Interface Name: emac_rx_reset Description: GMII/MII receive reset source synchronous to clk_rx_i from HPS			
Signal	**Width**	**Direction**	**Description**
rst_rx_n_o	1	Input	GMII/MII receive reset source from HPS. Active low reset.

Table 453. emac_rx_clk_in

Interface Name: emac_rx_clk_in Description: GMII/MII receive clock to HPS			
Signal	**Width**	**Direction**	**Description**
clk_rx_i	1	Output	GMII/MII receive clock to HPS

Table 454. emac_tx_clk_in

Interface Name: emac_tx_clk_in Description: GMII/MII transmit clock to HPS			
Signal	**Width**	**Direction**	**Description**
clk_tx_i	1	Output	GMII/MII transmit clock to HPS

Table 455. hps_gmii

Interface Name: hps_gmii Description: GMII/MII interface facing FPGA fabric			
Signal	**Width**	**Direction**	**Description**
mac_tx_clk_o	1	Output	GMII/MII transmit clock from HPS
mac_tx_clk_i	1	Input	GMII/MII transmit clock to HPS

continued...

Interface Name: hps_gmii Description: GMII/MII interface facing FPGA fabric			
Signal	**Width**	**Direction**	**Description**
mac_rx_clk	1	Input	GMII/MII receive clock to HPS
mac_rst_tx_n	1	Output	GMII/MII transmit reset source from HPS
mac_rst_rx_n	1	Output	GMII/MII receive reset source from HPS
mac_txd	8	Output	GMII/MII transmit data from HPS
mac_txen	1	Output	GMII/MII transmit enable from HPS
mac_txer	1	Output	GMII/MII transmit error from HPS
mac_rxdv	1	Input	GMII/MII receive data valid to HPS
mac_rxer	1	Input	GMII/MII receive data error to HPS
mac_rxd	8	Input	GMII/MII receive data to HPS
mac_col	1	Input	GMII/MII collision detect to HPS
mac_crs	1	Input	GMII/MII carrier sense to HPS
mac_speed	2	Output	MAC speed indication from HPS

Table 456. ptp

Interface Name: ptp Description: PTP interface facing FPGA fabric			
Signal	**Width**	**Direction**	**Description**
ptp_pps_out	1	Output	PTP pulse per second to FPGA soft logic
ptp_aux_ts_trig_in	1	Input	PTP auxiliary timestamp trigger from FPGA soft logic
ptp_tstmp_data_out	1	Output	PTP timestamp data from HPS to FPGA soft logic
ptp_tstmp_en_out	1	Output	PTP timestamp enable from HPS to FPGA soft logic

Table 457. mdio

Interface Name: mdio Description: MDIO interface facing PHY device			
Signal	**Width**	**Direction**	**Description**
mdo_out	1	Output	MDIO data output to FPGA bidirectional I/O buffer
mdo_out_en	1	Output	MDIO data output enable to FPGA bidirectional I/O buffer
mdi_in	1	Input	MDIO data input from FPGA bidirectional I/O buffer

Related Information

Avalon-MM Agent Interface on page 562
For more information about the Avalon-MM Agent interface, refer to the Avalon-MM Agent interface section.

48.6.1.4. Register

48.6.1.4.1. Register Memory Map

This register block exists only when the selected device is Arria V or Cyclone V. Each address offset represents one word of memory address space.

Name	Address Offset	Width	Attribute	Description
CTRL	0x0	2	R/W	Control Register

48.6.1.4.2. Register Description

Control Register

Table 458. Control Registers

Bit	Fields	Access	Default Value	Description
31:2	Reserved	N/A	0x0	Reserved
1:0	MAC_SPEED	R/W	0x0	This field indicates the speed mode used by HPS EMAC and PHY device. HPS software is required to write to this field once it has set the MAC Speed in the HPS EMAC register space after the auto-negotiation process. 0x0-0x1: 1000 Mbps (GMII) 0x2: 10 Mbps (MII) 0x3: 100 Mbps (MII)

48.6.1.5. Avalon-MM Agent Interface

The following information describes the characteristics of the Avalon agent interface of the HPS EMAC interface splitter core:

- Burst width: 32-bit
- Burst support: No
- Fixed read and write wait time: 0 cycle
- Fixed read latency: 1 cycle
- Lock support: No

48.7. Intel FPGA GMII to RGMII Converter Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2019.08.16	19.2	Added a note to clarify the device support for *Intel FPGA GMII to RGMII Converter Core*.
2018.05.07	18.0	Implemented editorial enhancements.

Date	Version	Changes
November 2015	2015.11.06	• Updated "Altera HPS EMAC Interface Splitter Core Interface" PTP table • Updated "Unsupported Features"
July 2014	2014.07.24	Initial release

49. Intel FPGA MII to RMII Converter Core

This chapter describes functionality of the Intel FPGA Media Independent Interface (MII) to Reduced Media Independent Interface (RMII) Converter core. RMII is a low pin count alternative to the MII for connecting a PHY to an Ethernet MAC.

The MII has 16 pins per port for data and control (4-bit), while the RMII has 8 pins. Therefore, the RMII is optimized for use in high-port density interconnect.

The RMII has the following characteristics:

- Supports 10 Mbps and 100 Mbps data rates.
- A single 50 MHz clock reference is sourced from the PHY or externally.
- Provides independent 2-bit wide (di-bit) transmit and receive data paths.
- Uses TTL signal levels, compatible with common digital CMOS ASIC process.

49.1. Supported Features

The Intel FPGA MII to RMII Converter core supports the following features:

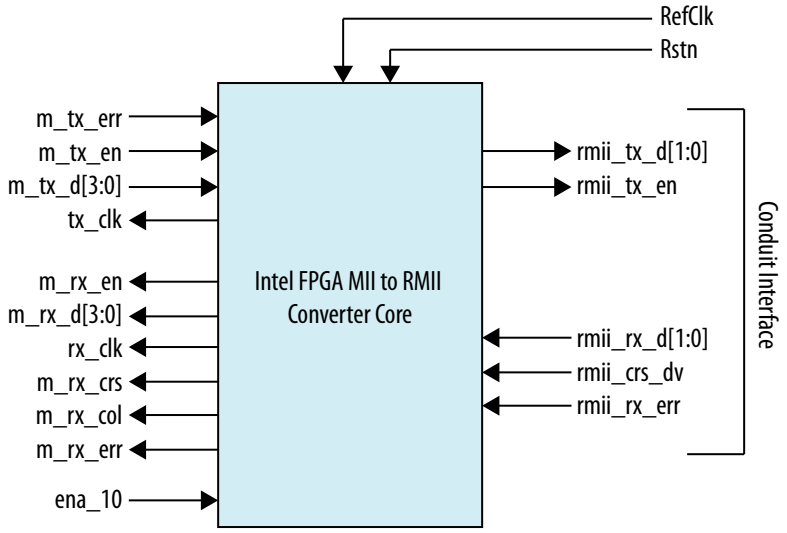
- Parameter based selection between 100Mbps and 10Mbps speed mode.
- Automatic detection of the receive throughput.
- Parameter based selection for RMII 1.2.
- Support error reporting from transmit MAC.
- Fixed frequency of 50 MHz.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

49.2. Interface Signals

Figure 143. Interface Signals



Signal Name	Type	Width (bit)	Description
Clock and Reset			
RefClk	Input	1	A fixed frequency interface clock of 50 MHz. It is driven by PHY.
Rstn	Input	1	Active low reset signal. EMAC provides this signal and it is connected to the MII to RMII Converter core and the PHY.
RMII: Core to RMII PHY Transmit Interface			
rmii_tx_d[1:0]	Output	2	Transmit data.
rmii_tx_en	Output	1	Transmit enable.
RMII: PHY to RMII Receive Interface			
rmii_crs_dv	Input	1	Multiplexed Carrier Sense or Data Valid Signal (RMII 1.2).
rmii_rx_d[1:0]	Input	2	Receive data.
rmii_rx_err	Input	1	Receive error.
MII: RMII Core to MAC Transmit Interface			
tx_clk	Output	1	Transmit clock: • 25 MHz in 100 Mbps speed mode • 2.5 MHz in 10 Mbps speed mode
m_tx_en	Input	1	Transmit data valid synchronized signal to transmit clock.
m_tx_d[3:0]	Input	4	Transmit data.
m_tx_err	Input	1	Transmit error.
MII: MAC to RMII Core Receive Interface			
rx_clk	Output	1	Receive clock.
m_rx_en	Output	1	Receive data valid (extracted from CRS_DV).

continued...

Signal Name	Type	Width (bit)	Description
m_rx_err	Output	1	Receive error.
m_rx_crs	Output	1	Ethernet carrier sense signal.
m_rx_col	Output	1	Ethernet collision signal.
m_rx_d[3:0]	Output	4	Receive data.
Core Speed Selection			
ena_10	Input	1	Indication that the MAC is configured to 10 Mbps throughput.

49.3. Parameters

Table 459. Parameters

Parameter	Description
MBPS	Set to: • **1**: for 100 Mbps speed mode • **0**: for 10 Mbps speed mode
THROUGHPUT CONTROL	When set to **1**, the IP core is set to fixed throughput, which is indicated by the MBPS parameter.
MAC_SPEED	When set to **1**, the MAC provides speed to the IP core.

Table 460. Parameter Usage Scenario

MAC_SPEED	MBPS	THROUGHPUT CONTROL	Scenario
1	N/A	N/A	RX and TX speed is provided by the input signal MIIMAC2CORESPEED10. If MIIMAC2CORESPEED10 == 1, the throughput is fixed at 10 Mbps, else it is fixed at 100 Mbps.
0	0	0	RX at dynamic speed detection. TX at fixed 10 Mbps speed.
0	0	1	RX at fixed 10 Mbps. TX at fixed 10 Mbps.
1	1	0	RX at dynamic speed detection. TX at fixed 100 Mbps speed.
1	1	1	RX at fixed 100 Mbps. TX at fixed 100 Mbps.

49.4. Functional Description

49.4.1. Clocking Scheme

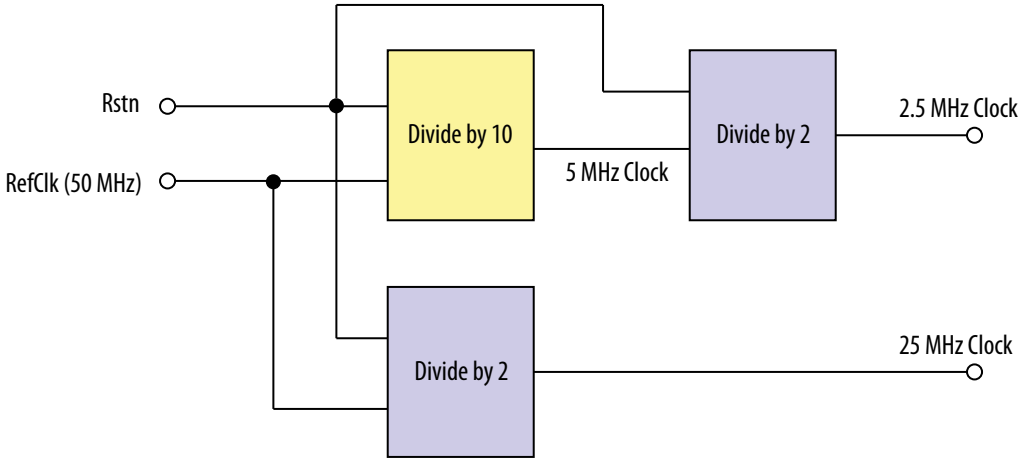
Either the PHY or an external source supplies a fixed 50 MHz reference clock to the IP core. The IP core instantiates two internal clock dividers which generates the required clocks for transmitter and receiver to support 100 Mbps and 10Mbps speed modes.

There are two division by 2-clock divider and one division by 10-clock divider.

Clock requirements:

- 25 MHz for IP core to MII MAC transmit clock for 100 Mbps data rate
- 2.5 MHz for IP core to MII MAC transmit clock for 10 Mbps data rate
- 5 MHz for RMII PHY to IP core receive clock for 10 Mbps data rate

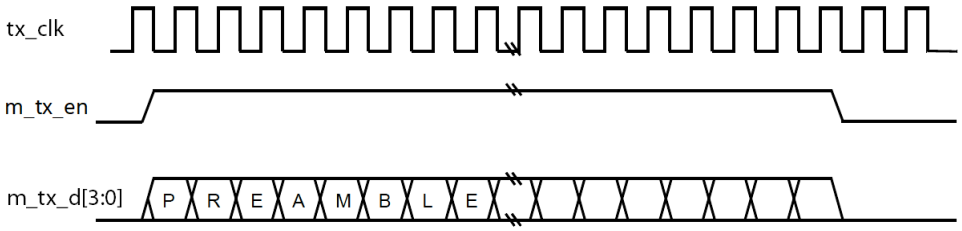
Figure 144. Clocking Scheme



49.4.2. Transmit Interface

The transmit logic takes 4-bit (nibble) data through the MII transmit interface from the MAC and transmits the same through the RMII interface to a RMII PHY. The following figure shows the timing relations between the different signals in a MII TX frame.

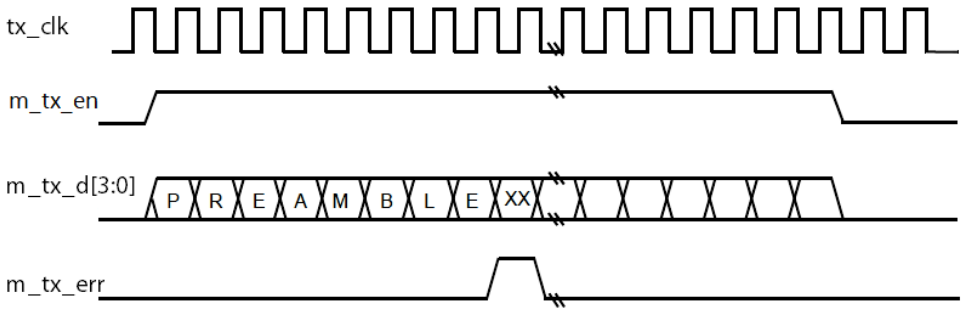
Figure 145. MII TX Frame



The parameter **MBPS** determines data rate of the transmit interface. If **MBPS** is set to 1, the TX speed in 100 Mbps; else if the **MBPS** is set to 0, the TX speed is 10 Mbps. The frequency of the TX clock is 25 Mhz in 100 Mbps mode and 2.5Mhz in 10Mbps mode. The IP core provides TX clock to MII MAC.

During transmission, the MII MAC asserts the `m_tx_en` signal and transmit 4-bit of data at each cycle of the TX clock. When there is an error in the transmission, the MII MAC asserts the `m_tx_err` signal synchronous to TX clock. The following figure shows a TX frame with `m_tx_err`.

Figure 146. TX Frame with Error Signal

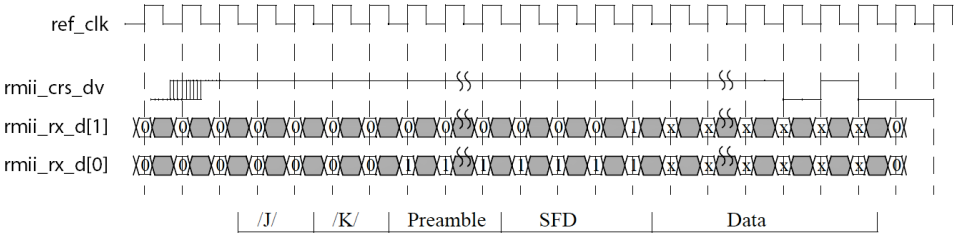


On detecting an TX error, the core transmits **01** pattern for the rest of the transmission over the output RMII interface, so that the frame fails the cyclic redundancy check (CRC).

49.4.3. Receive Interface

The receive interface is a 2-bit data interface which takes di-bits from the RMII MAC and accumulates 4-bits and sends it over the MII data interface to the MII MAC. The following figure shows a RMII receive transfer.

Figure 147. RMII Receive Transfer



The PHY asserts the Carrier Sense or Receive Data Valid (`rmii_crs_dv`) signal when the receive medium is non-idle. The `rmii_crs_dv` is de-asserted synchronously if the carrier is lost, but only at nibble boundaries. If the PHY has additional bits to be presented on `rmii_rx_d[1:0]` following the loss of carrier, the PHY asserts the `rmii_crs_dv` on cycles of `ref_clk` which represent the second di-bit of each nibble.

At the nibble boundary, the `rmii_crs_dv` toggles at 25 MHz in 100 Mbps mode and 2.5 MHz in 10 Mbps mode on the event of carrier loss. This helps the IP core to recover the receive data valid and carrier sense signal.

49.5. Intel FPGA MII to RMII Converter Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2020.07.22	20.2	Corrected the following signal names: • `ref_clk` to `RefClk` • `rst_n` to `Rstn` • `m_tx_dv` to `m_tx_en`
2019.12.16	20.1	Initial release.

50. Intel FPGA HPS GMII to TSE 1000BASE-X/SGMII PCS Bridge Core

50.1. Core Overview

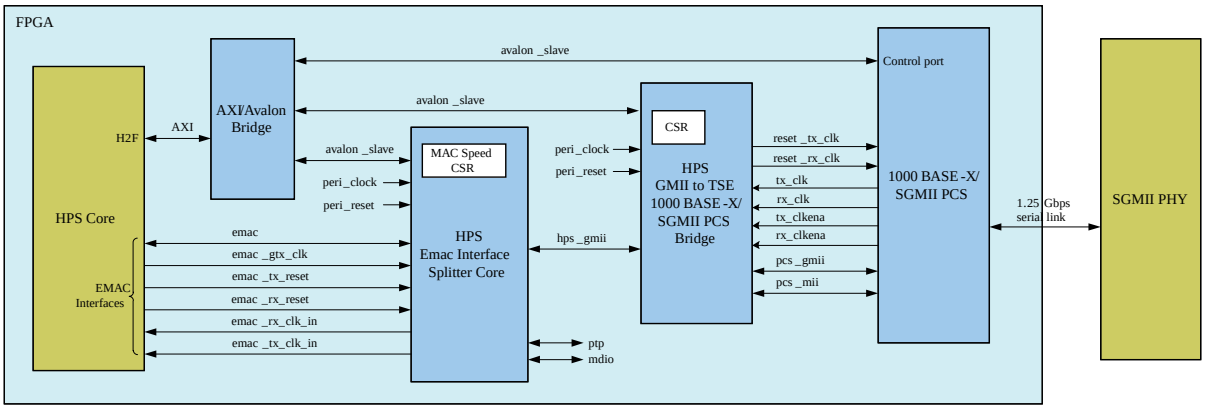
The Intel Hard Processor System (HPS) provides an Ethernet MAC function through its EMAC peripherals. The EMAC peripherals provide an RGMII or RMII interface to the HPS dedicated I/O or an GMII/MII interface to the FPGA I/O. For Serial Gigabit Media Independent Interface (SGMII), it is supported through the GMII/MII interface to FPGA fabric.

The Intel HPS GMII to TSE 1000BASE-X/SGMII PCS bridge is a soft IP core in FPGA fabric which provides logic to hook up the HPS's EMAC GMII/MII to the 1000BASE-X/SGMII PCS core for SGMII interface realization.

Note: This IP core supports Intel eASIC N5X devices.

50.2. Feature Description

Figure 148. Intel FPGA HPS GMII to TSE 1000BASE-X/SGMII PCS Bridge Core Block Diagram



The Intel FPGA HPS GMII to TSE 1000BASE-X/SGMII PCS bridge is not directly connected to the HPS component. Instead an intermediate component called the Altera HPS EMAC Interface Splitter core is used as a bridge between HPS core and Intel FPGA HPS GMII to TSE 1000BASE-X/SGMII PCS bridge. The intermediate component is responsible to split the EMAC conduit interface output from HPS core into several interfaces according to their function (`hps_gmii`, PTP, MDIO interfaces). It also responsible to manage differences between EMAC interfaces of the Arria V HPS, Cyclone V HPS, and Intel Arria 10 HPS.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Related Information

Intel FPGA HPS EMAC Interface Splitter Core on page 556

50.2.1. Supported Features

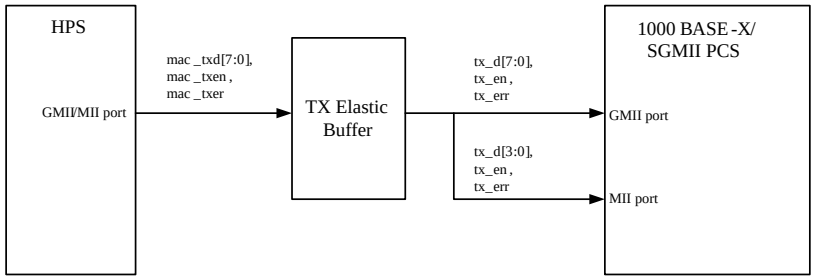
Features supported by the core:

- Enable HPS's EMAC GMII/MII connection Intel FPGA 1000BASE-X/SGMII PCS core
- Tri-speed (10/100/1000 Mbps) operation
- Dynamic speed switching

50.3. Core Architecture

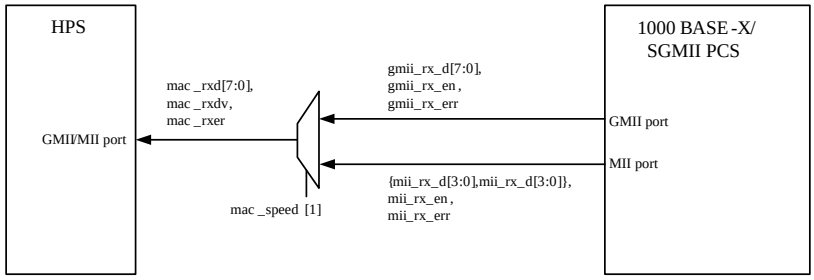
50.3.1. Data Path

Figure 149. Transmit Data Path



For transmit path, the GMII/MII data from the HPS goes through the transmit elastic buffer before going into the PCS GMII and MII port. The transmit elastic buffer is responsible for handling slight frequency differences between the transmit clock from HPS and the transmit clock generated from the PCS's block.

Figure 150. Receive Data Path



The PCS block has separate GMII and MII ports while the HPS has only single GMII and MII ports. Therefore a mux is needed in the receive data path. During MII mode, the 4 bits MII receive data bus is duplicated in order to feed 8 bits to the GMII/MII receive data bus of HPS. The `mac_speed` information from the HPS or from the CSR in Intel FPGA HPS EMAC Interface Splitter core is used as mux select.

50.3.2. Clock Scheme

During GMII mode (1000 Mbps), both the HPS and PCS blocks generate a transmit clock. The GMII/MII data from the HPS is synchronous to the HPS's internal PLL while the PCS block expects transmit data to be synchronous to its own transmit clock. To solve two different transmit clocks in a transmit data path, an elastic buffer is used for transmit data transmission.

During MII mode (10/100 Mbps), a transmit clock only comes from the PCS block. The transmit clock connected to the HPS is the gated version of transmit clock sent from PCS block with the worst duty cycle of 1% (high: 4 ns, low: 396 ns).

The receive clock comes from the PCS block. The receive clock connected to the HPS is the gated version of receive clock sent from the PCS block with worst duty cycle of 1% (high: 4 ns, low: 396 ns).

50.3.3. MAC Speed

Mac speed information is used to select different transmit clock sources.

Arria V or Cyclone V HPS cores do not provide mac speed information to the FPGA fabric. Therefore a control register is defined in the Intel FPGA HPS EMAC Interface Splitter core for software to configure it correctly according to the speed used by HPS EMAC and PHY device.

The Intel Arria 10 HPS provides mac speed information to the FPGA fabric. The control register in the Intel FPGA HPS EMAC Interface Splitter core is automatically removed.

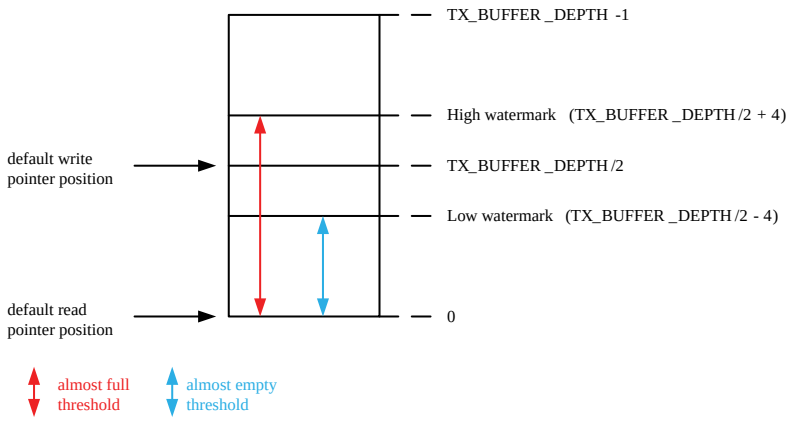
The two incoming `mac_speed` bits going into Intel FPGA HPS GMII to TSE 1000BASE-X/SGMII PCS bridge is treated as asynchronous and static. Only 1 bit (`mac_speed[1]`]) is being used to determine whether the MAC is operating in GMII or MII mode. Therefore a double synchronizer is enough to synchronize (`mac_speed[1]`). No additional filtering logic is needed unless both bits are used.

50.3.4. Transmit Elastic Buffer

Transmit elastic buffer is asynchronous FIFO with fix high and low watermark level. High watermark level is fixed at (TX_BUFFER_DEPTH/2 + 4) valid entries and low watermark level is fixed at (TX_BUFFER_DEPTH/2 - 4) valid entries. When the number of valid entries is equal or above the high watermark level, any IDLE symbol appearing at the write port of the buffer is dropped (IDLE symbol deletion). When the number of valid entries is equal or below low watermarks level, any IDLE symbol read from the read port does not increment the read pointer of the buffer (IDLE symbol insertion). IDLE symbol definition is `tx_en` = 0 and `tx_err` = 0.

The buffer depth is fixed at 64. Higher buffer depths provides more margins to avoid overflow/underflow condition to occur. The buffer depth configuration depends on the maximum Ethernet packet or maximum IDLE symbol separation in the Ethernet protocol and the maximum ppm different between the two clock sources.

Figure 151. Elastic Buffer Watermark Level



50.3.4.1. GMII to MII Mode Transition

The elastic buffer works when both write and read ports are having an equal or similar clock frequency. Ethernet operation allows dynamic speed mode changes. During a GMII to MII or MII to GMII mode transition, there could be a possibility that the transmit clock from HPS clock source and PCS block are out of sync in terms of clock frequency. If they are out of sync this causes an overflow/underflow condition to occur.

For example, during GMII to MII mode transition, the transmit clock from the PCS could be running at 25/2.5MHz while clock switching in HPS may yet to be completed and running at 125MHz. Clock switching in the PCS and HPS could incur a short period of an inactive clock as well due to graceful clock mux implementation. This challenge is handled through software.A register bit which act as a soft reset to the buffer is defined in this adapter core. Software is responsible to ensure the buffer is disabled when there is a change in the speed configuration of the PCS and MAC. The buffer is enabled only when configuration in both PCS and MAC blocks are completed and a valid transmit clock is running at both read and write ports of the buffer.

50.3.5. Avalon-MM Agent Interface

The following are the configuration of the Avalon-MM agent interface:

- Bus width: 32-bit
- Fixed read and write wait time: 0 cycles
- No burst support
- No lock support

50.3.6. Programming Model

Software is required to disable and enable the transmit data path accordingly whenever there is change in speed mode configuration.

In the case of Cyclone V and Arria V SoC devices, software is required to program the `mac_speed` register in Intel FPGA HPS EMAC Interface Splitter core as per MAC or PHY device setting.

Refer to the *Triple-Speed Ethernet IP Core User Guide* for programming sequence of the MAC and PCS block respectively.

Related Information

Triple-Speed Ethernet MegaCore Function User Guide

50.4. Configuration Parameters

50.5. Interface

Figure 152. Intel FPGA HPS GMII to TSE 1000BASE-X/SGMII PCS Bridge Top Level Interfaces

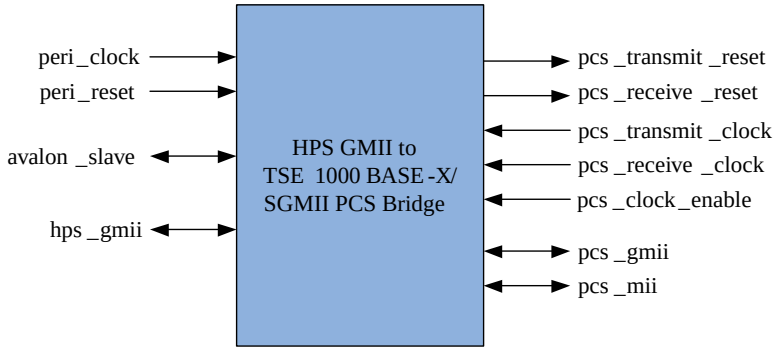


Table 461. Top Level I/O Port List

Signal	Width	Direction	Description
Interface Name: `peri_clock` Description: Peripheral clock interface			
`clk`	1	Input	Peripheral clock source
Interface Name: `peri_reset` Description: Peripheral reset interface			
`rst_n`	1	Input	Active low peripheral asynchronous reset source. This signal is asynchronously asserted and synchronously de-asserted. The synchronous de-assertion must be provided external to this core.
Interface Name: `avalon_slave` Description: Avalon MM agent interface for CSR access of this core			
`addr`	1	Input	Avalon-MM address bus. The address bus is in the unit of word addressing.
`read`	1	Input	Avalon-MM read control
`write`	1	Input	Avalon-MM write control
`writedata`	32	Input	Avalon-MM write data bus
`readdata`	32	Output	Avalon-MM read data bus
Interface name: `hps_gmii` Description: Conduit interface connected to HPS EMAC GMII/MII interface			
`mac_tx_clk_o`	1	Input	GMII/MII transmit clock from HPS

continued...

Signal	Width	Direction	Description
mac_tx_clk_i	1	Output	GMII/MII transmit clock to HPS
mac_rx_clk	1	Output	GMII/MII receive clock to HPS
mac_rst_tx_n	1	Input	GMII/MII transmit reset source from HPS. Active low reset.
mac_rst_rx_n	1	Input	GMII/MII receive reset source from HPS. Active low reset.
mac_txd	8	Input	GMII/MII transmit data from HPS
mac_txen	1	Input	GMII/MII transmit enable from HPS
mac_txer	1	Input	GMII/MII transmit error from HPS
mac_rxdv	1	Output	GMII/MII receive data valid to HPS
mac_rxer	1	Output	GMII/MII receive data error to HPS
mac_rxd	8	Output	GMII/MII receive data to HPS
mac_col	1	Output	GMII/MII collision detect to HPS
mac_crs	1	Output	GMII/MII carrier sense to HPS
mac_speed	2	Input	MAC speed indication from HPS
Interface name: pcs_transmit_reset Description: Transmit reset source from HPS			
pcs_rst_tx	1	Output	Inverted version of mac_rst_tx_n. Active high reset.
Interface name: pcs_receive_reset Description: Receive reset source from HPS			
pcs_rst_rx	1	Output	Inverted version of mac_rst_rx_n. Active high reset.
Interface name: pcs_transmit_clock Description: Transmit clock from PCS block			
pcs_tx_clk	1	Input	Transmit clock from PCS block.
Interface name: pcs_receive_clock Description: Receive clock from PCS block			
pcs_rx_clk	1	Input	Receive clock from PCS block
Interface name: pcs_clock_enable Description: Transmit and receive clock enabler from PCS block			
pcs_txclk_ena	1	Input	Transmit clock enabler from PCS block. This signal enables the pcs_tx_clk.
pcs_rxclk_ena	1	Input	Receive clock enabler from PCS block. This signal enables the pcs_rx_clk.
Interface name: pcs_gmii Description: GMII interface to the PCS block			
pcs_gmii_rx_dv	1	Input	Receive data valid from PCS block
pcs_gmii_rx_d	8	Input	Receive data from PCS block
pcs_gmii_rx_err	1	Input	Receive data error from PCS block
pcs_gmii_tx_en	1	Output	Transmit data enable to PCS block
pcs_gmii_tx_d	8	Output	Transmit data to PCS block

continued...

Signal	Width	Direction	Description
pcs_gmii_tx_err	1	Output	Transmit data error to PCS block
Interface name: pcs_mii Description: MII interface to the PCS block			
pcs_mii_rx_dv	1	Input	Receive data valid from PCS block
pcs_mii_rx_d	4	Input	Receive data from PCS block
pcs_mii_rx_err	1	Input	Receive data error from PCS block
pcs_mii_tx_en	1	Output	Transmit data enable to PCS block
pcs_mii_tx_d	4	Output	Transmit data to PCS block
pcs_mii_tx_err	1	Output	Transmit data error to PCS block
pcs_mii_col	1	Input	Collision detect from PCS block
pcs_mii_crs	1	Input	Carrier sense from PCS block

50.6. Registers

50.6.1. Register Memory Map

Each address offset represent one word of memory address space.

Table 462. Control Register Memory Map

Register	Address Offset	Width	Attribute
CTRL	0x0	1	R/W

50.6.2. Register Description

Table 463. Control Register (CTRL)

Bit	Fields	Access	Default Value	Description
31:1	Reserved	N/A	0x0	Reserved
0	TX_DISABLE	R/W	0x1	Transmit data path disable bit. This field disables the transmit data path of the adapter core. It acts as software reset to all transmit sequential logic in the adapter core (example the elastic buffer controller). 1: Transmit data path is disabled (default). 0: Transmit data path is enabled.

50.7. Intel FPGA HPS GMII to TSE 1000BASE-X/SGMII PCS Bridge Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2021.03.29	21.1	Added support for Intel eASIC N5X devices.
2018.05.07	18.0	Implemented editorial enhancements.
2017.05.08	17.0	Initial release.

51. Intel FPGA HPS EMAC to Multi-rate PHY GMII Adapter Core

Intel FPGA Hard Processor System (HPS) supports EMAC peripherals that provide RGMII or RMII interface to HPS dedicated IO or GMII/MII interface to FPGA IO with 8-bit data width.

On the other hand, the 1G/2.5G/5G/10G Multi-rate Ethernet PHY Intel FPGA IP core implements the IEEE 802.3.2005 standard which only supports 16-bit GMII/MII interface.

Intel FPGA HPS EMAC to Multi-rate PHY GMII Adapter Core act as an extension to provide logic adaption for the GMII/MII interface between HPS's EMAC and Multi-rate Ethernet PHY to enable Serial Gigabit Media Independent Interface (SGMII) realization.

51.1. Supported Features

Intel FPGA HPS EMAC to Multi-rate PHY GMII Adapter core supports the following features:

- Tri-speed (10/100/1000 Mbps) operation
- Dynamic speed switching
- Full duplex mode

Note: This IP core is only available in the Intel Quartus Prime Pro Edition.

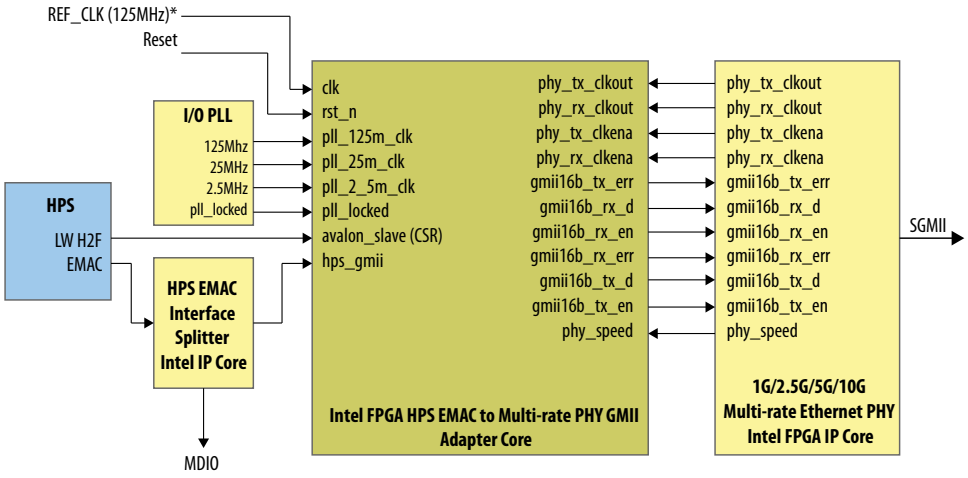
51.2. Functional Description

The following diagram illustrates the system level connection of Intel FPGA HPS EMAC to Multi-rate PHY GMII Adapter core.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Figure 153. Block Diagram



* Same as the reference clock for Multi-rate PHY IP core

The I/O PLL derives three clocks with frequency 125MHz, 25MHz, and 2.5MHz. The HPS EMAC Interface Splitter Intel IP core splits the EMAC conduit interface output from HPS into several interface signals according to their functions.

The Ethernet operation allows dynamic speed mode changes. During GMII to MII or MII to GMII mode transition, there is a possibility that the transmit clock from HPS clock source and PCS block are out of sync in terms of clock frequency which may cause an overflow/underflow situation. To avoid an overflow/underflow situation, use the TX_DISABLE register bit to disable the datapath during the change in speed configuration of the PCS and MAC.

Table 464. Avalon Memory-Mapped Interface Agent Interface

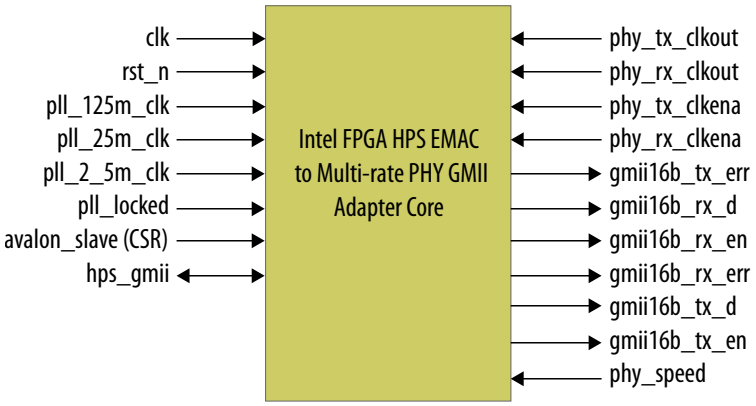
Configuration	Values
Bus width	32-bit
Burst support	No
Fixed read and write wait time	0 cycle
Fixed read latency	1 cycle
Lock support	No

Related Information

- RocketBoards.org: Intel Stratix 10 SoC SGMII Reference Design v18.1 - User Manual
- Intel FPGA HPS EMAC Interface Splitter Core on page 556

51.3. Interface Signals

Figure 154. Interface Signals



Signal	Width	Direction	Description
Peripheral Clock Interface (peri_clock)			
clk	1	Input	Peripheral clock source
Peripheral Reset Interface (peri_reset)			
rst_n	1	Input	Active-low peripheral asynchronous reset source. This signal is asynchronously asserted and synchronously de-asserted. The synchronous de-assertion must be provided external to this core.
PLL Clock Interface (pll_derived_clock_in)			
pll_125m_clk	1	Input	125MHz derived clock for adapter operation.
pll_25m_clk	1	Input	25MHz derived clock for adapter operation.
pll_2_5m_clk	1	Input	2.5MHz derived clock for adapter operation.
pll_locked	1	Input	PLL locked for adapter operation.
Avalon MM Agent Interface for CSR Access (avalon_slave)			
addr	1	Input	Avalon Memory-Mapped Interface address bus. The address bus is in the unit of Word addressing.
read	1	Input	Avalon Memory-Mapped Interface read control.
write	1	Input	Avalon Memory-Mapped Interface write control.
writedata	32	Input	Avalon Memory-Mapped Interface write data bus.
readdata	32	Output	Avalon Memory-Mapped Interface read data bus.
HPS EMAC GMII/MII Interface (hps_gmii)			
mac_tx_clk_o	1	Input	GMII/MII transmit clock from HPS.
mac_tx_clk_i	1	Output	GMII/MII transmit clock to HPS.
mac_rx_clk	1	Output	GMII/MII receive clock to HPS.
mac_rst_tx_n	1	Input	GMII/MII transmit reset source from HPS. Active low reset.
mac_rst_rx_n	1	Input	GMII/MII receive reset source from HPS. Active low reset.
mac_txd	8	Input	GMII/MII transmit data from HPS.

continued...

Signal	Width	Direction	Description
mac_txen	1	Input	GMII/MII transmit enable from HPS.
mac_txer	1	Input	GMII/MII transmit error from HPS.
mac_rxdv	1	Output	GMII/MII receive data valid to HPS.
mac_rxer	1	Output	GMII/MII receive data error to HPS.
mac_rxd	8	Output	GMII/MII receive data to HPS.
mac_col	1	Output	GMII/MII collision detect to HPS.
mac_crs	1	Output	GMII/MII carrier sense to HPS.
mac_speed	2	Input	MAC speed indication from HPS.
Multirate PHY GMII Clock Interface			
phy_rx_clkout	1	Input	Multirate PHY RX Clock Out.
phy_rx_clkena	1	Input	Multirate PHY RX Clock Enable.
phy_tx_clkout	1	Input	Multirate PHY TX Clock Out.
phy_tx_clkena	1	Input	Multirate PHY TX Clock Enable.
Multirate PHY GMII Data Interface (16-bit)			
gmii16b_rx_d	16	Input	Multirate PHY GMII receive data.
gmii16b_rx_dv	2	Input	Multirate PHY GMII receive data valid.
gmii16b_rx_err	2	Input	Multirate PHY GMII receive data error.
gmii16b_tx_d	16	Output	Multirate PHY GMII transmit data.
gmii16b_tx_en	2	Output	Multirate PHY GMII transmit data enable.
gmii16b_tx_err	2	Output	Multirate PHY GMII transmit data error.
Multirate PHY Speed Interface			
phy_speed	3	Input	Multirate PHY Operating Speed.

51.4. Register Memory Map and Description

Table 465. Register Memory Map

Each address offset represents one word of memory address space.

Name	Offset	Width	Access	Description
CTRL	0x0	1	R/W	Control Register

Table 466. Register Description

Bit	Fields	Access	Default Value	Description
Control Register (CTRL)				
31:1	Reserved	-	0x0	Reserved.
0	TX_DISABLE	R/W	0x1	Transmit data path disable bit. This field disables the transmit data path of the adapter core. It acts as software reset to all transmit sequential logic in the adapter core (for example the elastic buffer controller).

continued...

Bit	Fields	Access	Default Value	Description
				• 1: Transmit data path is disabled (default) • 0: Transmit data path is enabled .

51.5. Intel FPGA HPS EMAC to Multi-rate PHY GMII Adapter Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2019.04.01	19.1	Initial release.

52. Intel FPGA MSI to GIC Generator Core

52.1. Core Overview

In the PCI subsystem, Message Signaled Interrupts (MSI) is a feature that enables a device function to request service by writing a system-specified data value to a system-specified message address (using a PCI DWORD memory write transaction). System software initializes the message address and message data during device configuration, allocating one or more system-specified data and system-specified message addresses to each MSI capable function.

A MSI target (receiver), Intel FPGA PCIe RootPort Hard IP, receives MSI interrupts through the Avalon Streaming (Avalon-ST) RX TLP of type MWr. For Avalon-MM based PCIe RootPort Hard IP, the RP_Master issues a write transaction with the system-specified message data value to the system-specified message address of a MSI TLP received. This memory mapped mechanism does not issue any interrupt output to host the processor; and it relies on the host processor to poll the value changes at the system-specified message address in order to acknowledge the interrupt request and service the MSI interrupt. This polling mechanism may overwhelm the processor cycles and it is not efficient.

The Intel FPGA MSI-to-GIC Generator is introduced with the purpose of allowing level interrupt generation to the host processor upon arrival of a MSI interrupt. It exists as a separate module to Intel FPGA PCIe HIP for completing the interrupt generation to host the processor upon arrival of a MSI TLP.

52.2. Background

The existing implementation of the MSI target at Intel FPGA PCIe RootPort translates the MSI TLP received into a write transaction via PCIe Hard IP Avalon-MM Host port (RP_Master). No interrupt output directed to the host processor to kick start the service routine for the MSI sender is needed.

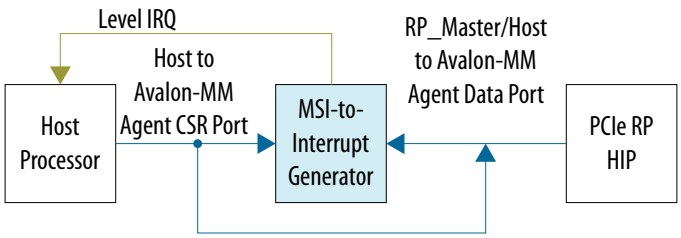
52.3. Feature Description

The Intel FPGA MSI-to-GIC Generator provides storage for the MSI system-specified data value. It also generates level interrupt output when there is an unread entry. The following figure illustrates the connection of the MSI-to-GIC Generator module in a PCIe subsystem.

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered

Figure 155. MSI-to-GIC Generator in PCIe RP system



This module is connected to RP_Master of PCIe RootPort HIP issuing memory map write transaction upon MSI TLP arrival. System-specified data value carried by the MSI TLP is written into the module storage. The same Avalon MM Data Agent port also connects to the host processor for MSI data retrieval upon interrupt assertion. An Intel FPGA MSI-to-GIC Generator module could contain data storage from one to 32 words of continuous address span. Each data word of storage is associated with a corresponding numbered bit of Status Bits and Mask Bits registers. Each data word address location can store up to 32 entries.

There is an up to 32-bit Status Register that indicates which storage word location has an unread entry. Also, there is a similar bit size of Interrupt Mask Register that is in place to allow control of module behavior by the host processor. The Interrupt Mask register provides flexibility for the host processor to disregard the incoming interrupt.

The base address assigned for Intel FPGA MSI-to-GIC Generator module in the subsystem should cover the system-specified message address of MSI capable functions during device configuration. Multiple Intel FPGA MSI-to-GIC Generator modules could be instantiated in a subsystem to cover different system-specified message addresses.

Avalon-MM Agent interfaces of this module honors fixed latency of access to ensure the connected host (in this case, the RP_Master) can successfully write into the module without back pressure. This avoids the PCIe upstream traffic from impact because of backpressuring of RP_Master.

Since MSI is multiple messages capable and multiple vectors are supported by each MSI capable function, there is a tendency that a system-specified message address receives more than one MSI message data before the host processor is able to service the MSI request. The Component is configurable to have each data word address to receive up to 32 entries, before any data value is retrieved. When you reach the maximum data value entry of 32, subsequent write transactions are dropped and logged. This ensures every write transaction to the storage has no back pressure which may lead to system lock up.

52.3.1. Interrupt Servicing Process

When a new message data is written into Intel FPGA MSI-to-GIC Generator module, the storage word associated Status bit is set automatically and a level interrupt output is then fired. The host processor that receives this interrupt output is required to service the MSI request, as indicated in the following procedure:

1. The host processor reads the Status Register to recognize which data word location of its storage is causing the interrupt.
2. The host processor reads the firing data word location for its system-specified message data value sent by the MSI capable function. Upon reading the data word, message data is considered consumed, the associated Status bit is then unset automatically. If the word location entry is empty, then the Status bit still remains asserted.
3. The host processor services either the MSI sender or the function who calls for the MSI.
4. Upon completing the interrupt service for the first entry, the host processor may continue to service the remaining entry if there is any residing inside the word location, by observing the associated Status bit.
5. The host processor may run through the Status Register and service each firing Status bit in any order.

52.3.2. Registers of Component

The following table illustrates the Intel FPGA MSI-to-GIC Generator registers map as observed by the host processor from its Avalon-MM CSR interfaces. The bit size of each register is numbered according to the configured number of data word storage for MSI message of the component. The maximum width of each register should be 32 bits because the configurable value range is from 1 to 32.

Table 467. CRA registers map

Word Address Offset	Register/ Queue Name	Attribute
0x0	Status register	R
0x1	Error register	RW *Note:* Write '1' to clear
0x2	Interrupt Mask register	RW

52.3.2.1. Status Register

The status register contains individual bits representing each of the data words location entry status. An unread entry sets the Status bit. The Status bit is cleared automatically when entry is empty. The value of the register is defaulted to ‘0’ upon reset.

The following table illustrates the Status register field.

Table 468. Status Register fields

Field Name	Bit Location
Status bit for message data word location [31:0]	31:0

52.3.2.2. Error Register

The Error register bit is set automatically only when the associated message data word location that contains the write entry, indicating it was dropped due to maximum entry limit reached. The Error bit indicates the possibility of the MSI TLP targeting the associated system-specified address. This condition should not happen as each MSI capable function is only allowed to send up to 32 MSI even with multiple vector supported.

The Error bit can be cleared by the host processor by writing '1' to the location.

Upon reset, the default value of the Error register bits are set to '0'.

The following table illustrates the Pending register field.

Table 469. Error Register fields

Field Name	Bit Location
Error bit for message data word location [31:0]	31:0

52.3.2.3. Interrupt Mask Register

The Interrupt Mask register provides a masking bit to individual Status bit before the Status is used to generate level interrupt output. Having the masking bit set, disregards the corresponding Status bit from causing interrupt output.

Upon reset, the default value of Interrupt Mask register is 0, which means every single data word address location is disabled for interrupt generation. To enable interrupt generation from a dedicated message entry location, the associated Mask bit needs to be set to '1'.

The following table illustrates the Interrupt Mask register field.

Table 470. Interrupt Mask Register fields

Field Name	Bit Location
Masking bit for Status [31:0]	31:0

52.3.3. Unsupported Feature

The message data entry Avalon-MM Agent represents the system-specified address for MSI function. The offset seen by MSI function should be similar to the offset seen by the host processors. As this Avalon-MM Agent interface is accessible (write and read) by both the host processor and the PCIe RP HIP, any read transaction to the offset address (system-specified address) is considered to have the message data entry consumed. Observing this limitation, only host host, which is expected to serve the MSI should read from the Avalon-MM Agent interface. A read from the PCIe RP_Master to the Avalon-MM Agent is prohibited.

52.4. Intel FPGA MSI to GIC Generator Core Revision History

Document Version	Intel Quartus Prime Version	Changes
2018.05.07	18.0	Implemented editorial enhancements.
2014.07.24	14.0	Initial release.