16-nm HBM PHY Firmware/Programming

User Guide

Version 0.10
April 19, 2018

pub-005556

For a comprehensive list of changes to this document, see the Revision History.

Corporate Headquarters	Website
San Jose, CA	www.broadcom.com

Table of Contents

16-nm HBM PHY Firmware/Programming User Guide 4
1 SBus Master Spico Firmware 4
1.1 Firmware Upload 4
1.2 Verilog Example 5
2 Spico Interrupts 7
2.1 Issuing Interrupts 7
2.2 Verilog Example 8
2.3 HBM Interrupts 9
2.4 HBM Spico Parameters 11
2.5 Error Codes 17
2.6 Operation Codes 19
3 HBM Firmware Operations 20
4 Recommended Reset and Initialization Procedure 25
4.1 PHY Configuration and HBM Mode Registers 25
4.2 HBM Mode Registers 26
4.3 CKE 27
5 Direct Control Without Firmware 27
5.1 HBM/PHY Reset 27
5.2 IEEE 1500 Access 28
5.3 IEEE 1500 TMRS 33
6 Recommended Verilog Test Protocol 35
7 Revision History 36
7.1 Version 0.10, April 19, 2018 36
7.2 Version 0.9. December 1, 2017 36
7.3 Version 0.8. August 1, 2017 36
7.4 Version 0.7, April 18, 2017 36
7.5 Version 0.6, December 22, 2017 37
7.6 Version 0.5, November 9, 2016 37
7.7 Version 0.4, August 15, 2016 37
7.8 Version 0.3, May 31, 2016 37
7.9 Version 0.2, April 22, 2016 37
7.10 Version 0.1, March 2016 37

16-nm HBM PHY Firmware/Programming User Guide

The Broadcom® High Bandwidth Memory (HBM) PHY interface is a self-contained, hardened macro that provides the logic and I/O interface for connection to a second-generation High Bandwidth Memory (HBM2) device. Configuration and test features are accessed through IEEE 1500 interfaces contained within the PHY and the HBM device. The PHY contains an SBus receiver that provides read/write control of both IEEE 1500 interfaces.

By providing access to these features via the SBus, multiple methods of control are available. The SBus Master block that drives the SBus ring can be controlled by the TAP or core interfaces. Embedded within the SBus Master is a Spico microprocessor that can also control the devices connected on the SBus ring. When used with the firmware provided by Broadcom, HBM-related operations can be automated for improved simplicity and speed.

Configuration and test capabilities accessible via the IEEE 1500/SBus interface include:

- Program PHY configuration and timing parameters
- Reset of the PHY and HBM
- Apply HBM hard lane repairs as soft repairs in the PHY
- Check interposer lanes for faults and apply soft or hard repairs to correct errors
- Perform address word and data word AC lane testing
- Perform PHY self-loopback and internal loopback testing
- Read HBM temperature via the IEEE 1500 interface
- HBM vendor-specific MBIST testing

This user guide describes the programming interface for the HBM PHY, firmware usage, and programming guidelines.

1 SBus Master Spico Firmware

Spico firmware automates various functional and verification tasks. The HBM PHY does not have a dedicated Spico microprocessor. Instead, it uses the Spico embedded within the SBus Master macro. For performance reasons, a unique, dedicated SBus Master macro must be used for each HBM interface. While reducing the round-trip time for SBus operations, having one SBus ring per interface also allows each interface to be run in parallel. The firmware is designed to only support a single interface.

The HBM firmware is not provided as a hardware ROM, but only as software ROM files that must be uploaded to the SBus Master prior to running HBM operations.

Two builds of HBM firmware are provided. One build supports all HBM core functionality except for MBIST operations. A separate build is provided to support just MBIST operations. Each build works for all vendors and HBM devices. The correct firmware files are:

sbus_master.hbm.0x####_2002.rom (HBM core)

sbus_master.hbm.0x####_2012.rom (HBM MBIST)

where 0x#### is the revision and 2002 and 2012 are the build IDs written in hex.

1.1 Firmware Upload

Before running any of the firmware-based interrupts and procedures described in this document, the Spico microprocessor must run through its start-up sequence and the firmware must be uploaded. These procedures are documented in the *Broadcom 16 nm SBus Master Specification* and the *Broadcom 16 nm Spico Firmware User Guide*. To verify that the firmware was properly uploaded, the firmware revision and build ID should be queried.

1.2 Verilog Example

The following Verilog tasks describe the upload procedure. Two tasks are provided. The first tasks demonstrates how firmware should be uploaded in a production system. Because this method can be slow in simulation, a faster task is provided to upload the firmware directly into the SBus Master Spico memory. This method can only be used for RTL simulations.

```
task upload_sbus_master_firmware_sbus;
  input [0:256*8-1] rom_file_vl;
  integer fd;                   // File Descriptor
  integer word_count;                // numbe of words in the burst
  integer addr;                 // address
  integer valid;                // Returns 1 for each word read.  Returns -1 for
end-of-file.
  reg [31:0] sbus_data;         // Addr/Data sent via sbus to load rom
  reg [9:0] imem_data;          // data word read from file

  $display( "INFO: %t Uploading Sbus Master Spico ROM", $time );

  // Place SPICO into Reset and Enable off
  sbus_driver.write(32'hfd, 1, 32'hc0);

  // Remove Reset, Enable off, IMEM_CNTL_EN on
  sbus_driver.write(32'hfd, 1, 32'h0240);

  // Set starting IMEM address for burst download
  sbus_driver.write(32'hfd, 3, 32'h80000000);

  // Fast Load the ROM code
  addr = 0;
  valid = 1;
  fd = $fopen(rom_file_vl, "r");

  while (valid > 0) begin
    sbus_data = 0;
    word_count = 0;

    // data word 1
    valid = $fscanf(fd, "%h", imem_data);
    if (valid  > 0) begin
      sbus_data = imem_data & 'h3ff;
      word_count += 1;
      valid = $fscanf(fd, "%h", imem_data);
    end

    // data word 2
    if (valid  > 0) begin
      sbus_data = sbus_data | ((imem_data & 'h3ff) << 10);
      word_count += 1;
      valid = $fscanf(fd, "%h", imem_data);
    end

    // data word 3
    if (valid  > 0) begin
```

```
        sbus_data = sbus_data | ((imem_data & 'h3ff) << 20);
        word_count += 1;
      end

      // Word count
      if (word_count  > 0) begin
        sbus_data = sbus_data | (word_count << 30);

        // Send the instruction burst
        sbus_driver.write(32'hfd, 32'h14, sbus_data);
        addr += word_count;
      end
    end
    $fclose(fd);

    sbus_driver.write(32'hfd, 32'h14, 32'hc000); // Pad with 0's
    sbus_driver.write(32'hfd, 32'h14, 32'hc000); // Pad with 0's
    sbus_driver.write(32'hfd, 32'h14, 32'hc000); // Pad with 0's

    sbus_driver.write(32'hfd, 32'h01, 32'h40); // Set IMEM_CNTL_EN off
    sbus_driver.write(32'hfd, 32'h01, 32'h140); // Set SPICO_ENABLE on

    // Give Spico time to get through init code
    $display( "INFO: %t Waiting for spico to initialize", $time );
    `ifdef RTL
      wait( `SBUS_MASTER_PATH.spico.spico_main.spico_state == 'h12 );
      repeat (10) @(posedge `SBUS_MASTER_PATH.spico_clk);
    `else
      repeat (1000) @(posedge `SBUS_MASTER_PATH.spico_clk);
    `endif
  endtask : upload_sbus_master_firmware_sbus



  task upload_sbus_master_firmware_direct;
    input [0:256*8-1] rom_file_vl;

    $display( "INFO: %t Uploading Sbus Master Spico ROM", $time );

    // ADDR   1       core_clk   ADDR_1
    // REG    0        0     RW    IGNORE_BROADCAST
    // REG    1        0     RW    IDCODE_OVERRIDE_GATE
    // REG    5:2      0     RW    REV_IDCODE_CNTL
    // REG    6        0     RW    SBUS_CNTL_GATE
    // REG    7        1     RW    RESET
    // REG    8        0     RW    ENABLE
    // REG    9        0     RW    IMEM_CNTL_EN
    // REG    10       0     RW    DMEM_CNTL_EN
    // REG    11       0     RW    DMEM_EXTENDED_CNTL_EN

    // Open Gate and insure that memories and Spico are in Reset

    // IMEM_CNTL_EN=0, SBUS_CNTL_GATE=1, RESET=1
    sbus_driver.write(32'hfd, 1, 32'hc0);
    // Release Reset (allows memories to operate) and make sure
    //interrupt bit is low

    // IMEM_CNTL_EN=0, SBUS_CNTL_GATE=1, RESET=1
    sbus_driver.write(32'hfd, 1, 32'h40);

    // Fast Load the ROM code
    `SBUS_MASTER_PATH.load_imem ( rom_file_vl );

    // Set IMEM_CNTL_EN off
    sbus_driver.write(32'hfd, 32'h01, 32'h40);

    // Set SPICO_ENABLE on
    sbus_driver.write(32'hfd, 32'h01, 32'h140);

    // Give Spico time to get through init code
    $display( "INFO: %t Waiting for spico to initialize", $time );
    wait( `SBUS_MASTER_PATH.spico.spico_main.spico_state == 'h12 );
    repeat (10) @(posedge `SBUS_MASTER_PATH.spico_clk);

    $display( "INFO: %t Spico to initialized", $time );
  endtask : upload_sbus_master_firmware_direct
```

2 Spico Interrupts

The HBM firmware functionality is controlled by interrupts issued to the Spico microprocessor using SBus commands.

2.1 Issuing Interrupts

To issue an interrupt, the interrupt code and data are set, followed by raising the interrupt bit. The interrupt result must then be read. Some interrupts take time to process before a valid result is returned. While processing, the busy bit of the output result is set high. When this bit is cleared, the result is valid and both result data and status can be read.

Table 1 shows a partial SBus address map for the SBus Master Spico processor describing the interrupt registers.

Table 1 Partial SBus Address Map

Address	Register	Default	Access	Description
2	15:0	0	RW	SPICO_INTERRUPT_CODE
	31:16	0	RW	SPICO_INTERRUPT_DATA
7	0	0	RW1C	INTERRUPT
8	14:0	0	R	SPICO_INTERRUPT_STATUS
	15	0	R	SPICO_INTERRUPT_BUSY
	31:16	0	R	SPICO_INTERRUPT_RESULT

Spico interrupts are fully documented in the *Broadcom Spico Firmware User Guide*.

2.2 Verilog Example

The following is example Verilog code to issue a Spico interrupt and retrieve the results.

```
task sbus_master_spico_interrupt;
  input [15:0] interrupt_code;
  input [15:0] interrupt_value;
  output [32:0] data;
  output error;
  integer timeout;

  begin
    // Set spico interrupt code and value
    sbus_write(32'hfd, 2, ((interrupt_value << 16) | interrupt_code));

    // Assert Interrupt
    sbus_read(32'hfd, 7, data);
    sbus_write(32'hfd, 7, (data | 1));

    // Lower Interrupt
    sbus_write(32'hfd, 7, (data & 32'hfffffffe));

    // Wait for interrupt to complete
    timeout = 100;
    $display("INFO: Waiting for interrupt 0x%02x to complete", interrupt_code);

  while (timeout > 0) begin
      sbus_read(32'hfd, 8, data);
      $display("INFO: Interrupt 0x%02x result/status: 0x%04x / 0x%04x",
        interrupt_code, (data >> 16), (data & 16'hffff) );
     // Check if interrupt busy flag is set
     if ((data & 32'h8000) == 0)
       break;
     end

    error = 0;
    if (timeout == 0)
    begin
      $display( "ERROR: Interrupt 0x%02x timeout while waiting for completion.",
      interrupt_code );
      error = 1;
    end
    else
    begin
      // Read again for since the read is un-triggered, this
      // means the data read when the status changed to "done"
      // may not be valid. So read again to get a valid result
      //data.
      sbus_read(32'hfd, 8, data);
      data = (data >> 16) & 32'hffff;
    end
  end
endtask
```

2.3 HBM Interrupts

Table 2 lists the HBM-specific interrupt codes that are defined for the HBM firmware.

Table 2 HBM Firmware Interrupt Codes

Operations	Description
0x30	Launch operation on all channels
0x31	Launch operation on a single channel
0x32	Get operation result
0x33	Get parameter value
0x34	Set parameter
0x35	Set parameter value

0x30 – Launch operation on all channels

Input	data[7:0]	Operation number to perform
Return Value	0x1	Successful start of test
	0x3ff	An HBM operation is currently in progress

Description This interrupt launches various HBM-related operations, such as HBM/PHY reset, lane repair, and HBM verification. Table 7 lists all the operations supported by firmware. Each operation is numbered. The requested operation is run on all channels for the operations that are channel specific. Some operations, such as reading the HBM Device ID, are not channel-specific.

The interrupt immediately returns with a 0x1 if successfully launched. Once launched, the operation continues to completion. 0x3ff is returned when an operation is currently in progress and the requested operation cannot be launched.

Some operations might take a significant amount of time to run. Regardless of the length of the operation, interrupt 0x32 must be polled to check if the operation has completed and to check if errors were detected.

Some firmware operations can be performed while in mission mode because they do not affect the operation of the HBM or PHY. Other operations require the PHY channel to be placed in test mode and the memory controller to halt mission mode. To ensure that this event occurs, a PHY update handshake is performed. This handshake sequence is outlined below.

1. Firmware operation interrupt is issued.
2. Channel specific DFI_PHYLVL_REQ_N signals are asserted from the PHY.
3. Memory controller halts operation and asserts DFI_PHYLVL_ACK_N.
4. PHY de-asserts DFI_INIT_COMPLETE and enters test mode.
5. Firmware completes requested operation.
6. PHY enters mission mode, asserts DFI_INIT_COMPLETE, de-asserts DFI_PHYLVL_REQ_N.
7. Memory controller resumes mission mode.

The PHY update handshake can be bypassed by setting the hbm_ignore_phyupd_handshake Spico parameter to 1. Also note that the PHY never asserts DFI_PHYLVL_REQ_N unless a firmware operation is requested by the user. If the memory controller has not been initialized and does not respond to the handshake at power-on, the handshake must be bypassed.

Operations that run in test mode must be launched using interrupt 0x32, which enables the operation on all channels. This is because a PHY/HBM reset is performed when entering test mode, which affects all channels. Operations that do not require test mode can use interrupt 0x31to run on a single channel when it is supported. Table 7 lists the operations that run in test mode and in mission mode. Because test mode operations perform a reset automatically, a separate reset operation is not required.

0x31 – Launch operation on a single channel

Input data[7:0] Operation number to perform

data[15:8] Channel to run

Return Value 0x1 Successful start of test

0x3ff An HBM operation is currently in progress

Description This interrupts works like interrupt 0x30 except that it allows for running operations on a single channel. Multiple channels cannot be selected. See 0x30 – Launch operation on all channels for further details.

Only operations that run in mission mode can be launched on a single channel. Operations that run in test mode must be run on all channels because a PHY/HBM reset is performed upon entering test mode, which resets all channels. Table 7 lists the operations that run in test mode and in mission mode.

0x32 – Get operation result

Input Result type to read

Return Value Result data

Description This interrupt reads various status and result values. The value passed to the interrupt indicates which result to read. Some results are common regardless of the operation that was launched, while others might be operation-specific.

Table 3 Result Types and Descriptions

Result Type	Result Description
0x00	Operation Status: Bit[0] – Operation done Bit[1] – Operation active Bit[2] – Operation errors detected
0x01	Global error code
0x02	Channel 0 error code
0x03	Channel 1 error code
0x04	Channel 2 error code
0x05	Channel 3 error code
0x06	Channel 4 error code
0x07	Channel 5 error code
0x08	Channel 6 error code
0x09	Channel 7 error code
0x0a	Channel 0 last operation
0x0b	Channel 1 last operation
0x0c	Channel 2 last operation
0x0d	Channel 3 last operation
0x0e	Channel 4 last operation
0x0f	Channel 5 last operation
0x10	Channel 6 last operation
0x11	Channel 7 last operation
0x12–0x29	hbm_spare_0 through hbm_spare_23

While an operation is in progress (launched with interrupt 0x30), the status can be checked by reading result 0x00. Bit[1] is high while the operation is running. When completed, bit[0] goes high and bit[1] goes low. If any errors are detected, bit[2] is set high.

When errors are detected, the global error code and the channel-specific error codes and operations should be read. When an error is not channel-specific, the global error code is set. An example might be a timeout error while waiting for the BIST state machine to run. For errors that are channel-specific, the error code for that channel is saved as well as the operation that was currently being run. When a collection of tests are being run, this is useful to identify which step failed.

Results 0x12 through 0x29 return hbm_spare_0 through hbm_spare_23. These results allow for up to 24 data results to be returned from an operation. The results will depend on the operation run but may include data, such a failing lanes, device temperature, device ID, and more. See individual operation descriptions for additional details.

0x33 – Get parameter value

Input Parameter offset to read

Return Value Parameter value

Description The HBM firmware has many user-configurable parameters. The current value of any parameter can be queried with this interrupt by providing the parameter offset to the interrupt. See Section 2.4, HBM Spico Parameters, for parameter names and descriptions.

0x34 – Set parameter offset

Input Parameter offset to write

Return Value

0x01	Success
0x3ff	Illegal offset

Description Setting an HBM parameter requires two interrupts. First, this interrupt specifies the parameter offset to program. Then, interrupt 0x35 must specify the parameter value and save it to memory.

0x35 – Set parameter value

Input Parameter value to write

Return Value

0x01	Success
0x3ff	Error

Description This interrupt is part of a two-step process to program a parameter value. First, interrupt 0x34 must specify the parameter offset. Then this interrupt can specify the parameter value and save the value to memory.

2.4 HBM Spico Parameters

The HBM firmware has a number of user configuration parameters that are used to control the behavior of each operation. These can be queried and set using interrupts 0x33, 0x34, and 0x35. Table 4 shows available parameters.

Table 4 HBM Spico Parameters

Parameter Offset	Parameter Name	Default Value
0x00	Reserved	0xff
0x01	hbm_max_timeout	0x3e8
0x02	hbm_tinit1_cycles	0x222e
0x03	hbm_tinit2_cycles	0x00
0x04	hbm_tinit3_cycles	0x5573
0x05	hbm_tinit4_cycles	0x00

Table 4 HBM Spico Parameters (Continued)

Parameter Offset	Parameter Name	Default Value
0x06	hbm_tinit5_cycles	0x09
0x07	hbm_rw_latency_offset	0x02
0x08	hbm_latency_odd_n_even	0x01
0x0a	hbm_mode_register0	0x03
0x0b	hbm_mode_register1	0x91
0x0c	hbm_mode_register2	0x96
0x0d	hbm_mode_register3	0xe1
0x0e	hbm_mode_register4	0x02
0x0f	hbm_mode_register5	0x00
0x10	hbm_mode_register6	0x64
0x11	hbm_mode_register7	0x02
0x12	hbm_mode_register8	0x00
0x13	hbm_phy_config0	0xa407
0x14	hbm_phy_config1	0x074c
0x15	hbm_phy_config2	0x4ca4
0x16	hbm_phy_config3	0x0798
0x17	hbm_phy_config4	0x4ca4
0x18	hbm_phy_config5	0xa407
0x19	hbm_phy_config6	0x004c
0x1a	hbm_lbp_drv_imp	0x00
0x1b	hbm_delay_config_dll	0x0
0x1c	hbm_ignore_phyupd_handshake	0x01
0x1d	hbm_tupdmrs_cycles	0x1f4
0x1e	hbm_t_rdlat_offset	0x04
0x1f	hbm_mbist_repair_mode	0x02
0x20	hbm_samsung_mbist_pattern	0x0
0x21	hbm_samsung_mbist_hard_repair_cycles	0x1964
0x22	hbm_hard_lane_repair_cycles	0x479a
0x24	hbm_mbist_bank_address_end	0x0f
0x25	hbm_mbist_row_address_end	0x3fff
0x26	hbm_mbist_column_address_end	0x3f
0x27	hbm_freq	0x7d0
0x28	hbm_div_mode	0x1
0x29	hbm_cke_exit_state	0x0
0x2a	hbm_test_mode_register0	0x73
0x2b	hbm_test_mode_register1	0x91
0x2c	hbm_test_mode_register2	0xa7
0x2d	hbm_test_mode_register3	0xa4
0x2e	hbm_test_mode_register4	0x03
0x2f	hbm_test_mode_register5	0x0
0x30	hbm_test_mode_register6	0x0

Table 4 HBM Spico Parameters (Continued)

Parameter Offset	Parameter Name	Default Value
0x31	hbm_test_mode_register7	0x2
0x32	hbm_test_mode_register8	0x0
0x33	hbm_ctc_run_cycles	0xff
0x34	hbm_ctc_channel_ignore	0x00
0x35	hbm_ctc_initial_address_lo	0x00
0x36	hbm_ctc_initial_address_hi	0x00
0x37	hbm_ctc_max_address_lo	0xffff
0x38	hbm_ctc_max_address_hi	0x7f
0x39	hbm_test_t_rdlat_offset	0x07
0x3a	hbm_mode_register15	0x00
0x3b	hbm_test_mode_register15	0x00
0x3c	hbm_bypass_testmode_reset	0x00
0x3d	hbm_disable_addr_lane_repair	0x00
0x3e	hbm_ctc_pattern_type	0x00
0x3f	hbm_bypass_repair_on_reset	0x00
0x40	hbm_stack_height	0x00
0x41	hbm_manufacturer_id	0x01
0x42	hbm_density	0x03
0x43	hbm_manually_configure_id	0x00
0x44	hbm_parity_latency	0x00
0x45	hbm_test_parity_latency	0x00
0x46	hbm_dfi_t_rddata_en	0x00
0x47	hbm_manually_config_nwl	0x00
0x48	hbm_ctc_pseudo_channel	0x03
0x49	hbm_ctc_sbref_en	0x4a
0x4a	hbm_ctc_retention_cycles	0xff
0x4b	hbm_ctc_user_data_seed	0x0
0x62	hbm_samsung_mbist_workaround	0x1
0x63	hbm_dword_op	0x3
0x66	hbm_model_number	0x0
0x67	hbm_date_code	0x0

0x01 – hbm_max_timeout – The number of polling cycles until a timeout error occurs. This value is used for all wait loops to prevent infinite loops.

0x02 – hbm_tinit1_cycles – HBM timing parameter for RESET_n signal LOW time at power-up (after stable power.) This value is specified in wait cycles where the total delay is 8 × spico_clk_period × cycles. The default is 200 µs for a 350 MHz Spico clock. For Verilog simulations, this value can be reduced to avoid long run times. The minimum allowed value for simulation depends on the specific HBM model used.

0x03 – hbm_tinit2_cycles – HBM timing parameter for CKE LOW time before RESET_n de-assertion. This value is specified in wait cycles where the total delay is 8 × spico_clk_period × cycles.

0x04 – hbm_tinit3_cycles – HBM timing parameter for CKE and WRST_n LOW time after RESET_n de-assertion. This value is specified in wait cycles where the total delay is 8 × spico_clk_period × cycles. The default is 500 µs for a 350 MHz Spico clock. For Verilog simulations, this value can be reduced to avoid long run times. The minimum allowed value for simulation depends on the specific HBM model used.

0x05 – hbm_tinit4_cycles – HBM timing parameter for stable clock before CKE HIGH.
This value is specified in wait cycles where the total delay is 8 × spico_clk_period × cycles.

0x06 – hbm_tinit5_cycles – HBM timing parameter for idle time before first MRS command.
This value is specified in wait cycles where the total delay is 8 × spico_clk_period × cycles. The default is 200 ns for a 350 MHz Spico clock. For Verilog simulations, this value can be reduced to avoid long run times. The minimum allowed value for simulation depends on the specific HBM model used.

0x07 – hbm_rw_latency_offset – PHY timing parameter used during dword testing. This parameter is auto-set during the test so users do not need to modify this parameter. The parameter value is determined based on the test type as well as the hbm_freq and hbm_div_mode parameters.

0x08 – hbm_latency_odd_n_even – PHY timing parameter used during dword testing. This parameter is auto-set during the test so that users do not need to modify this parameter. The parameter value is determined based on the test type as well as the hbm_freq and hbm_div_mode parameters.

0x0a – 0x12 - hbm_mode_register* – Values to program into the HBM mode registers.
These are programmed upon exiting test mode operations to enable mission mode.
Only 8-bits of each hbm_mode_register* are used.

0x13 – hbm_phy_config0 – Bits[15:0] of the PHY_CONFIG PHY IEEE 1500 configuration register.
The hbm_phy_config* parameters are used to program the PHY_CONFIG values after an HBM/PHY reset. The default values are the optimal values determined by Broadcom during silicon characterization. End users should not need to adjust these.

0x14 – hbm_phy_config1 – Bits[31:16] of the PHY_CONFIG PHY IEEE 1500 configuration register.

0x15 – hbm_phy_config2 – Bits[47:32] of the PHY_CONFIG PHY IEEE 1500 configuration register.

0x16 – hbm_phy_config3 – Bits[63:48] of the PHY_CONFIG PHY IEEE 1500 configuration register.

0x17 – hbm_phy_config4 – Bits[79:64] of the PHY_CONFIG PHY IEEE 1500 configuration register.

0x18 – hbm_phy_config5 – Bits[95:80] of the PHY_CONFIG PHY IEEE 1500 configuration register.

0x19 – hbm_phy_config6 – Bits[103:96] of the PHY_CONFIG PHY IEEE 1500 configuration register.

0x1a – hbm_lbp_drv_imp – Controls the driver impedance for the loopback pads.

0x1b – hbm_delay_config_dll – Configuration for the PHY DLL.

0x1c – hbm_ignore_phyupd_handshake – Setting this parameter to 1 disables the PHY_REQ_N/PHY_ACK_N handshake when entering test mode. This is the default behavior because test mode operations typically occur before the memory controller has been initialized and can respond to the handshake. This parameter can be set to 0 at any time to re-enable the handshake upon entering test mode.

0x1d – hbm_tupdmrs_cycles – HBM timing parameter for WDR update to Mode Register valid delay.
This value is specified in wait cycles where the total delay is 8 × spico_clk_period × cycles. The default is approximately 2 × 250 WDR clock cycles at 50 MHz.

0x1e – hbm_t_rdlat_offset – Override value for the T_RDLAT_OFFSET bits of the PHY_CONFIG when programming mission mode configuration. When programming PHY_CONFIG, this value is merged with those from hbm_phy_config*. This simplifies usage because only this one parameter can be adjusted while ignoring the rest of the PHY_CONFIG bits.

0x1f – hbm_mbist_repair_mode – Type of repair performed after an MBIST run:
0 – No repair
1 – Hard repair
2 – Soft repair

0x20 – hbm_mbist_pattern – MBIST pattern to run. This is specific to the type of HBM device:

Samsung:

0 – SCAN
1 – MARCH

SKH:

0x0 – YMC (replaces PAUSE) on all channels in parallel
0x1 – GROSS on all channels in parallel
0x2 – XMC on all channels in parallel
0x80 – YMC on all channels serially
0x81 – GROSS on all channels serially
0x82 – XMC on all channels serially

0x21 – hbm_samsung_mbist_hard_repair_cycles – Number of wait cycles for an individual bit cell repair.

0x22 – hbm_hard_lane_repair_cycles – Number of wait cycles for an individual hard lane repair.

0x24 – hbm_mbist_bank_address_end – Maximum bank address to use when running SK hynix MBIST.
This limits the length of th MBIST run during Verilog simulation.

0x25 – hbm_mbist_row_address_end – Maximum row address to use when running SK hynix MBIST.
This limits the length of th MBIST run during Verilog simulation.

0x26 – hbm_mbist_column_address_end – Maximum column address to use when running SK hynix MBIST.
This limits the length of th MBIST run during Verilog simulation.

0x27 – hbm_freq – The CLK_2X frequency in MHz.
This determines the optimal timing parameter values during test.

0x28 – hbm_div_mode – PHY DIV mode.
This determines the optimizing timing parameter values during test.

0 – DIV1 mode
1 – DIV2 mode

0x29 – hbm_cke_exit_state – Value to set CKE to after exiting test mode.
When running a test mode operation, the CKE can be left in a different state than when entering test mode.
This parameter sets the state of CKE upon exiting test mode.

0x2a–0x32 – hbm_test_mode_register* – MRS values during test operation.
Mode registers must be set differently during test mode than during mission mode. These are the optimal mode register values determined from silicon characterization for use in test mode.

0x33 – hbm_ctc_run_cycles – Number of cycles to wait for CTC to complete when running all CTC simultaneously.
Total delay is approximately 0x7ff8 × cycles × SBus period

0x34 – hbm_ctc_channel_ignore – Ignores the CTC results for the specified channels.
When running all CTC simultaneously, the results from specific channels are ignored. All channels still run, but a failing CTC does not cause the operation to fail. Each bit represents one channel. Setting the channel bit to 1 indicates the channel results should be ignored.

0x35 – hbm_ctc_initial_address_lo – Bits[15:0] of the initial read/write CTC address.

0x36 – hbm_ctc_initial_address_hi – Bits[31:16] of the initial read/write CTC address.

0x37 – hbm_ctc_max_address_lo – Bits[15:0] of the maximum read/write CTC address.

0x38 – hbm_ctc_max_address_hi – Bits[31:16] of the maximum read/write CTC address.

0x39 – hbm_test_t_rdlat_offset – Override value for the T_RDLAT_OFFSET bits of the PHY_CONFIG when programming test mode configuration. When programming PHY_CONFIG, this value is merged with those from hbm_phy_config*. This simplifies usage because only this one parameter can be adjusted while ignoring the rest of the PHY_CONFIG bits.

0x3a – hbm_mode_register15 – Value to program into the HBM mode register15. This value is programmed upon exiting test mode operations to enable mission mode.

0x3b – hbm_test_mode_register15 – Value to program into the HBM mode register15 during test mode operations.

0x3c – hbm_bypass_testmode_reset – All firmware operations that run in test mode perform a reset upon entering testmode. This is to ensure the test is run in a clean state and that it can clear some programming such as soft lane repairs. Setting this register to a 1 causes the reset to be skipped upon entering test mode.

0x3d – hbm_disable_addr_lane_repair – When set to 1, this skips checking for errors on the address lanes during the Lane Repair and Lane Verify operations.

0x3e – hbm_ctc_pattern_type – Selects the pattern type to run during CTC tests.

0 – Low power PRBS pattern. Performs alternating writes/reads with PRBS data.
1 – High power 0xaf50 pattern. Pattern is written to entire memory followed by continuous back-to-back reads.
2 – High power DBI pattern. Modified 0xaf50 pattern to cause DBI toggles and non-repeating transactions.
3 – High power PRBS. Similar to high power except data is 2048b of PRBS data.

0x3f – hbm_bypass_repair_on_reset – Bypass hard lane repair. When set to 1, HBM hard lane repairs will not be applied to the PHY as soft lane repairs.

0x40 – hbm_stack_height – Sets the stack height for the HBM device. This parameter is set automatically based on the DEVICE_ID unless the hbm_manually_configure_id parameter is set to 1. Valid values are:

0 – STACK_4HI
1 – STACK_8H

0x41 – hbm_manufacturing_id – Sets the manufacturing ID for the HBM device. This parameter is set automatically based on the DEVICE_ID unless the hbm_manually_configure_id parameter is set to 1. Valid values are:

1 – SAMSUNG
6 – SKH
15 – MICRON

0x42 – hbm_density – Sets the density for the HBM device. This parameter is set automatically based on the DEVICE_ID unless the hbm_manually_configure_id parameter is set to 1. Valid values are:

1 – DENSITY_1GB
2 – DENSITY_2GB
3 – DENSITY_4GB
4 – DENSITY_8GB
5 – DENSITY_16GB
6 – DENSITY_32GB

0x43 – manually_configure_id – Enables user configuration of HBM ID parameters. During firmware tests, the DEVICE_ID of the HBM may be read to determine the stack height, density, and manufacturer. When this parameter is set to 1, these values will be manually set from the defined hbm_stack_height, hbm_manufacturing_id, and hbm_density parameters.

0x44 – hbm_parity_latency – Sets the value of parity latency in the MRS and PHY_CONFIG registers for mission mode operation.

0x45 – hbm_test_parity_latency – Sets the value of parity latency in the MRS and PHY_CONFIG registers for test mode operation.

0x46 – hbm_dfi_t_rddata_en – Sets the value of the CFG_DFI_T_RDDATA_EN NWL CSR in the NWL memory controller during initialization. If the value is left at the default of 0, the value will be auto-determined based on memory controller type and HBM vendor.

0x47 – hbm_manually_configure_nwl – Allows for default NWL CSRs to be overwritten. When set to 1, the NWL CSR values are derived from the hbm_mode_register* parameters where possible. When set to 0, the production test values will be used.

0x48 – hbm_ctc_pseudo_channel – For dual-channel memory controllers, this parameter determines which pseudo-channel to start/stop CTC tests for. The default value is 0b11. Below are possible values.

0b00 – Illegal value
0b01 – Pseudo Channel 0
0b10 – Pseudo Channel 1
0b11 – Pseudo Channel 0/1

This is used with the start_ctc and stop_ctc operations allows multiple interrupts to be made to start/stop CTC operations of specific channels/psuedo-channels.

0x49 – hbm_ctc_sbref_en – Sets the value of the CFG_SBREF_EN CSR in the NWL memory controller during initialization. This value defaults to 0.

0x4a – hbm_ctc_retention_cycles – Adds wait cycles between the HBM writes and reads during the high-power CTC testing to allow for data retention testing. The default value is 0 which injects no wait cycles. The wait time can be determined with the follow equation: wait_time = 0xfff * hbm_ctc_retention_cycles * 13 * spico_clk_period.

0x4b – hbm_ctc_user_data_seed – Sets the seed value for generating random PRBS data during CTC testing.

0x62 – hbm_samsung_mbist_workaround - Enables/disables using the Samsung workarounds for MBIST. Different revision of the Samsung HBMs require workaround for to enable MBIST to properly work. The workarounds may be disable by setting this parameter to 0. The default is 1.

0x63 – hbm_dword_op – Indicates whether the DWORD operations should perform a read operation, write operation, or both. Valid settings are:

1 – WRITE
2 – READ
3 – WRITE and READ (default)

0x66 – hbm_model_number – Sets the model number of the HBM device. This parameter is set automatically based on the DEVICE_ID unless the hbm_manually_configure_id parameter is set to 1.

0x67 – hbm_date_code – Sets the date code of the HBM device. This parameter is set automatically based on the DEVICE_ID unless the hbm_manually_configure_id parameter is set to 1.

2.5 Error Codes

When an HBM operation runs, errors can occur. If so, the error codes for each channel can be read using interrupt 0x32. The following is a description of each error code.

Table 5 Possible Error Codes

Error Code	Error Name
0x01	ERROR_DETECTED
0x02	ERROR_UNEXPECED_RESET_STATE
0x03	ERROR_ILLEGAL_CHANNEL_NUMBER
0x04	ERROR_TIMEOUT_WAITING_FOR_1500_DONE
0x05	ERROR_TIMEOUT_WAITING_FOR_BIST_DONE
0x06	ERROR_DATA_COMPARE_FAILED
0x07	ERROR_ALL_CHANNEL_NOT_SELECTED_FOR_RESET
0x08	ERROR_REPAIR_LIMIT_EXCEEDED
0x09	ERROR_NON_REPAIRABLE_FAULTS_FOUND
0x0a	ERROR_MBIST_FAILED
0x0b	ERROR_EXCEEDED_BANK_REPAIR_LIMIT
0x0c	ERROR_ALL_CHANNELS_NOT_ENABLED
0x0d	ERROR_TIMEOUT_WAITING_FOR_PHYUPD_HANDSHAKE

Table 5 Possible Error Codes (Continued)

Error Code	Error Name
0x0e	ERROR_CHANNEL_UNREPAIRABLE
0x0f	ERROR_NO_FUSES_AVAILABLE_FOR_REPAIR
0x10	ERROR_TIMEOUT_WAITING_FOR_VALID_TEMP
0x11	ERROR_CHANNEL_FAILURES_EXIST
0x12	ERROR_UNKNOWN_ERROR
0x13	ERROR_TIMEOUT_WAITING_FOR_NWL_INIT
0x14	ERROR_CTC_WRITE_READ_COMPARE_FAILURE
0x15	ERROR_CTC_NO_WRITES_PERFORMED
0x16	ERROR_CTC_NO_READS_PERFORMED
0x17	ERROR_LANE_ERRORS_DETECTED
0x19	ERROR_UNSUPPORTED_HBM_CONFIGURATION
0x1a	ERROR_HBM_MISR_PRESET_FAILED
0x1b	ERROR_PHY_MISR_PRESET_FAILED
0x1c	ERROR_CORE_POWERON_RST_L_ASSERTED
0x1d	ERROR_UNSUPPORTED_INTERRUPT

0x01 – ERROR_DETECTED – Generic error code that indicates an error occurred.

0x02 – ERROR_UNEXPECTED_RESET_STATE – Indicates that the reset signals are not set properly. A reset must be performed.

0x03 – ERROR_ILLEGAL_CHANNEL_NUMBER_REQUESTED – A channel number outside the range of 0 to 7 was specified.

0x04 – ERROR_TIMEOUT_WAITING_FOR_1500_DONE – A timeout occurred performing the SBus-to-IEEE 1500 handshake.

0x05 – ERROR_TIMEOUT_WAITING_FOR_BIST_DONE – A timeout occurred while waiting for a BIST operation to complete.

0x06 – ERROR_DATA_COMPARE_FAILED – A data comparison failed during an operation.

0x07 – ERROR_ALL_CHANNEL_NOT_SELECTED_FOR_RESET – When performing reset operations, all channels must be selected because the resets apply to all channels.

0x08 – ERROR_LANE_REPAIR_LIMIT_EXCEEDED – One or more errors were detected that could not be repaired.

0x09 – ERROR_NON_REPAIRABLE_LANE FAULTS_FOUND – Errors were detected on lanes that are non-repairable.

0x0a – ERROR_MBIST_FAILED – The MBIST operation indicates a memory fault was detected.

0x0b – ERROR_EXCEEDED_BANK_REPAIR_LIMIT – More MBIST errors were detected within a bank than can be repaired.

0x0c – ERROR_ALL_CHANNELS_NOT_ENABLED – Indicates that the operation was requested to run on a single channel when the operation requires that all channels are enabled.

0x0d – ERROR_TIMEOUT_WAITING_FOR_PHYUPD_HANDSHAKE – No response received from the memory controller on the appropriate C*_DFI_PHYLVL_ACK_N ports after the firmware asserted C*_DFI_PHYLVL_REQ_N.

0x0e – ERROR_SKH_EXCEEDED_BANK_REPAIR_LIMIT – More MBIST errors were detected within a bank than can be repaired when running SK hynix MBIST.

0x0f – ERROR_NO_FUSES_AVAILABLE_FOR_REPAIR – Not enough fuses available for repair during SK hynix MBIST. This applies to soft and hard repair.

0x10 – ERROR_TIMEOUT_WAITING_FOR_VALID_TEMP – A timeout occurred while reading the temperature of the HBM device.

0x11 – ERROR_CHANNEL_FAILURES_EXIST – Indicates a channel-specific error occurred and the channel error codes need to be queried.

0x12 – ERROR_UNKNOWN_ERROR – A error was raised during the operation, but no error codes were set.

0x13 – ERROR_TIMEOUT_WAITING_FOR_NWL_INIT – A timeout occurred while waiting for the NWL memory controller to complete its initialization procedure.

0x14 – ERROR_CTC_WRITE_READ_COMPARE_FAILURE – A compare error was detected while running the CTC operation.

0x15 – ERROR_CTC_NO_WRITES_PERFORMED – Write operations were not performed during CTC operations.

0x16 – ERROR_CTC_NO_READS_PERFORMED – Read operations were not performed during CTC operations.

0x17 – ERROR_LANE_ERRORS_DETECTED – Indicates a lane error was detected during the Lane Verify operation.

0x19 – ERROR_UNSUPPORTED_HBM_CONFIGURATION – Memory controller programming does not support the HBM configuration read from the DEVICE_ID.

0x1a – ERROR_HBM_MISR_PRESET_FAILED – Indicates that the presetting of the HBM AWORD or DWORD MISR failed.

0x1b – ERROR_PHY_MISR_PRESET_FAILED – Indicates that the presetting of the PHY AWORD or DWORD MISR failed.

0x1c – ERROR_CORE_POWERON_RST_L_ASSERTED – Indicates that the CORE_POWERON_RST_L SBus register is low when entering test mode. This occurs when the HBM_POWER_ON_RESET_L and TEST__HBM_XGEN_DISABLE core ports are set low.

0x1d – ERROR_UNSUPPORTED_INTERRUPT – Indicates that the requested interrupt is not available in the current firmware build. This error would be seen when calling the MBIST interrupt with the core build or calling a core interrupt with the MBIST build.

2.6 Operation Codes

When an HBM operation runs, errors can occur. When the error occurs, the current operation code for the failing channel is saved along with the error code. Below is a list of possible operation codes that are saved.

Table 6 Channel Operation Codes

Operation Code	Operation Name
0x01	OP_BYPASS
0x02	OP_DEVICE_ID
0x03	OP_AWORD
0x04	OP_AERR
0x05	OP_AWORD_ILB
0x06	OP_AERR_ILB
0x07	OP_AERR_INJ_ILB
0x08	OP_DWORD_WRITE
0x09	OP_DWORD_READ
0x0a	OP_DERR
0x0b	OP_DWORD_UPPER_ILB
0x0c	OP_DWORD_LOWER_ILB

Table 6 Channel Operation Codes (Continued)

Operation Code	Operation Name
0x0d	OP_DERR_ILB
0x0e	OP_DERR_INJ_ILB
0x0f	OP_LANE_REPAIR
0x10	OP_DEVICE_TEMP
0x11	OP_CONNECTIVITY_CHECK
0x12	OP_RESET
0x13	OP_MBIST
0x14	OP_BITCELL_REPAIR
0x15	OP_AWORD_SLB
0x16	OP_AERR_SLB
0x17	OP_AERR_INJ_SLB
0x18	OP_DWORD_UPPER_SLB
0x19	OP_DWORD_LOWER_SLB
0x1a	OP_DERR_SLB
0x1b	OP_DERR_INJ_SLB
0x1c	OP_TMRS
0x1d	OP_CHIPPING
0x1e	OP_MC_INIT
0x1f	OP_CTC
0x20	OP_APPLY_LANE_REPAIR
0x21	OP_BURN_LANE_REPAIR
0x22	OP_CATTRIP

3 HBM Firmware Operations

The HBM firmware supports many operations. This section documents each operation and how to interact with it. Certain operations can be run in mission mode, while others must be executed while in test mode. During test mode, the BIST_EN SBus register is asserted, which allows the PHY to take control of the DFI interface.

Table 7 Operations and Descriptions

Operation Number	Operation Description	Mission Mode/Test Mode
0x00	Reset PHY and HBM	Test mode
0x01	Reset PHY FIFO Pointers	Mission mode
0x02	Reset HBM	Test mode
0x03	Package Test Flow	Test mode
0x06	Connectivity Check	Mission mode
0x07	Bypass Test	Mission mode
0x08	Read Device ID	Mission mode
0x09	AWORD Test	Test mode
0x0a	AERR Test	Test mode

Table 7 Operations and Descriptions (Continued)

Operation Number	Operation Description	Mission Mode/Test Mode
0x0b	DWORD Test	Test mode
0x0c	DERR Test	Test mode
0x0d	Lane Repair	Test mode
0x0e	AWORD ILB Test	Test mode
0x0f	DWORD ILB Test	Test mode
0x10	Read Device Temp	Mission mode
0x11	Burn Hard Lane Repairs	Test mode
0x12	Apply Hard Lane Repairs to PHY	Test mode
0x13	Run Samsung MBIST	Test mode
0x14	Run SK hynix MBIST	Test mode
0x15	Reset Mode Registers	Mission mode
0x16	Reset PHY_CONFIG	Mission mode
0x17	AWORD SLB Test	Test mode
0x18	DWORD SLB Test	Test mode
0x19	Chipping Test	Mission mode
0x1a	Reserved	Mission mode
0x1b	AERR ILB Test	Test mode
0x1c	AERR SLB Test	Test mode
0x1d	DERR ILB Test	Test mode
0x1e	DERR SLB Test	Test mode
0x1f	Initialize NWL Memory Controllers	Mission mode
0x20	Run CTCs	Mission mode
0x21	Lane Verify	Test mode
0x22	Start All CTCs	Mission mode
0x23	Stop All CTCs	Mission mode
0x24	Load TMRS Code	Mission mode
0x25	Release CTC Control	Mission mode
0x26	CATTRIP Test	Test mode

0x00 – Reset PHY and HBM – This operation performs a complete reset of the HBM and PHY circuitry, including main power-on resets, IEEE 1500 resets, and state machine resets. After reset, the HBM mode registers and PHY configuration registers are programmed to mission mode values as defined by Spico parameters. Additionally, hard lane repairs are read from the HBM and applied to the PHY as soft lane repairs. This operation applies to all channels. Due to the tINIT timing parameters required by the HBM during the reset sequence, this interrupt takes a long time to simulate. This time can be significantly reduced by modifying the hbm_tinit*_cycles Spico parameters, but only do so for simulation. This operation applies to all channels.

Approximate runtime: 760 µs

0x01 – Reset PHY FIFO Pointers – This operation resets only the PHY mission-mode FIFO pointers. The PHY IEEE 1500 registers are not affected. This operation applies to all channels.

Approximate runtime: 36 µs

0x02 – Reset HBM – Performs an asynchronous functional reset of the HBM using the HBM_RESET IEEE 1500 WDR. This operation does not perform an IEEE 1500 reset, but is equivalent to asserting the HBM RESET_n. The HBM mode registers are reprogrammed after the reset to ensure proper mission mode operation. This operation applies to all channels.

Approximate runtime: 50 µs

0x05 – Power-On Flow – Before running mission mode operations, the power-on flow can be run to reset the PHY and HBM as well as run lane AC lane testing. This will ensure that communication between the PHY and HBM is functional. The following operations are performed:

— HBM PHY Reset
— AWORD test
— DWORD test
— AERR test
— DERR test

Approximate runtime: 46.8 ms in Mode 2

0x06 – Connectivity Check – To ensure connectivity of HBM-related devices, SBus receivers are polled to ensure the HBM SBus ring contains one APC block, one PLL, eight or sixteen CTC blocks (depending on the flavor of CTC used), and eight STOP blocks. For each PHY channel, the ctc_id_out value is read to find the associated CTC block and check the CTC configuration.

Approximate runtime: TBD

0x07 – Bypass Test – This simple test verifies that the BYPASS IEEE 1500 WDR can be written to. This test can be used to verify IEEE 1500 operation.

Approximate runtime: 80 µs

0x08 – Device ID Test – Reads the HBM device ID value from the IEEE 1500 register. The 82-bit result is written to the PHY SBus SPARE registers.

DeviceID[31:0]SPARE_0
DeviceID[63:32] SPARE_1
DeviceID[81:64]SPARE_2

The result is also saved in the hbm_spare results access with interrupt 0x32. These are saved as follows:

hbm_spare_0 - device_id[15:0]
hbm_spare_1 - device_id[31:16]
hbm_spare_2 - device_id[47:32]
hbm_spare_3 - device_id[63:38]
hbm_spare_4 - device_id[79:64]
hbm_spare_5 - device_id[82:80]

Approximate runtime: 50 µs

0x09 – AWORD Test – Performs address word AC-testing from the PHY to HBM. Refer to the *Broadcom 16 nm HBM PHY Specification* for details.

Approximate runtime: 290 µs

0x0a – AERR Test – Performs AERR AC-testing from the PHY to HBM. Refer to the *Broadcom 16 nm HBM PHY Specification* for details.

Approximate runtime: 485 µs

0x0b – DWORD Test – Performs data word AC-testing between the PHY and HBM. Refer to the *Broadcom 16 nm HBM PHY Specification* for details.

Approximate runtime: 515 µs

0x0c – DERR Test – Performs DERR AC-testing between the PHY and HBM. Refer to the *Broadcom 16 nm HBM PHY Specification* for details.

Approximate runtime: 305 µs

0x0d – Lane Repair – Performs DC-testing of all lanes to identify lanes requiring repair. This action is done by running a series of vectors as defined by a custom implementation of the Wagner Algorithm. The vectors are scanned using the EXTEST_TX and EXTEST_RX IEEE 1500 WDRs. This algorithm detects shorts and opens. A similar technique is used for PC board boundary-scan testing. Note that lane repair ignores any hard lane repairs that may be been previously applied.

A unique signature is scanned on each lane with all lanes in a channel running simultaneously. The received signature is then analyzed. Depending on the received signature, the following error types can be detected:

— Shorts to VDD and GND

— Shorts to another signal

— Opens

For lanes with errors, the firmware first determines if the lane can be repaired because not all lanes have redundancy. If a lane without redundancy has an error, the operation fails. For repairable lanes, the firmware checks if a valid repair can be made and programs the SOFT_LANE_REPAIR WDR in the PHY and HBM appropriately. For DWORD repairs, a Mode 2 repair is made, if possible, keeping DBI functionality. If a Mode 2 repair cannot be made, a Mode 1 repair is attempted. If there are too many errors to repair, the operation fails.

Lane repair counts and repaired lane numbers are saved to the hbm_spare registers and access with interrupt 0x32. These are saved as follows:

hbm_spare[0] - Total number of lanes with faults
hbm_spare[1-23] - Lane number with fault

The results are saved regardless of whether the operation passes or fails. Each lane number can be mapped to a signal name by referring to the JEDEC HBM Spec's EXTEST_TX WDR definition. The lane repair values match this register definition.

Approximate runtime: 45.5 ms

0x0e – AWORD ILB Test – Performs address internal loopback testing for in-system diagnostic testing. Refer to the *Broadcom16 nm HBM PHY Specification* for details.

Approximate runtime: 85 µs

0x0f – DWORD ILB Test – Performs data internal loopback testing for in-system diagnostic testing. Refer to the *Broadcom 16 nm HBM PHY Specification* for details.

Approximate runtime: 172 µs

0x10 – Read Device Temperature – Reads the HBM temperature by way of the IEEE 1500 interface. The values are read until the VALID runtime goes high. The result is stored in the PHY SBus SPARE_0 register as well as the hbm_spare_0 result register accessed with interrupt 0x32.

Approximate runtime: 30 µs

0x11 – Burn Hard Lane Repairs – Burns lane repair fuses in the HBM to permanently apply soft lane repairs. Prior to running this operation, the lane repair operation can be run to create the soft repairs or soft repairs can be manually applied by writing to the SOFT_LANE_REPAIR IEEE 1500 WDR. This operation can be run multiple times.

0x12 – Apply Hard Lane Repairs as Phy Soft Lane Repairs – Reads the hard lane repair information from the HBM and applies them as soft repairs in the PHY. This is required to ensure that both the PHY and HBM are using the same repair codes. This operation is performed as part of the reset operation.

0x13 – Run MBIST – Runs the MBIST for either Samsung or SKH devices. The appropriate vendor specific MBIST to run will be auto-determined based on the DEVICE_ID. This operation replaces the previously SKH-specific MBIST operation 0x14.

The results of MBIST are used to program soft or hard bit cell repairs. Repairs can also be skipped. The repair mode is controlled by setting the hbm_mbist_repair_mode parameter. If repairs are made, a second MBIST run is performed to verify the integrity of the repair. If errors are still found, an error is issued.

When used with Samsung HBM devices, the HBM clock frequency (DRAM clock) must be run at 800 MHz. The frequency is not configured by the firmware and must be set prior to launching this firmware operation.

The number of MBIST repairs are saved to the hbm_spare_0 register accessed with interrupt 0x32. This register also can be accessed by reading the PHY Sbus SPARE_1 register.

0x14 – Run MBIST – This operation is now equivalent to operation 0x13.

0x15 – Reset Mode Registers – Programs the HBM mode registers using the values stored in the hbm_mode_register* firmware parameters.

0x16 – Reset PHY_CONFIG – Programs the PHY_CONFIG WDR using the values stored in the hbm_phy_config* firmware parameters.

0x17 – Run AWORD SLB – Performs address self-loopback testing for wafer diagnostic testing. Refer to the *Broadcom 16 nm HBM PHY Specification* for details.

Approximate runtime: 85 µs

0x18 – Run DWORD SLB – Performs data self-loopback testing for wafer diagnostic testing. Refer to the *Broadcom 16 nm HBM PHY Specification* for details.

Approximate runtime: 172 µs

0x19 – Chipping Test – Performs a base-die wafer chipping test. This is specific to Samsung devices.

0x1b – AERR ILB Test – Performs AERR internal-loopback testing for wafer diagnostic testing. Refer to *Broadcom16 nm HBM PHY Specification* for details.

0x1c – AERR SLB Test – Performs AERR self-loopback testing for wafer diagnostic testing. Refer to *Broadcom 16 nm HBM PHY Specification* for details.

0x1d – DERR ILB Test – Performs DERR internal-loopback testing for wafer diagnostic testing. Refer to *Broadcom 16 nm HBM PHY Specification* for details.

0x1e – DERR SLB Test – Performs DERR self-loopback testing for wafer diagnostic testing. Refer to *Broadcom 16 nm HBM PHY Specification* for details.

0x1f – Initialize NWL Memory Controllers – Initializes the NWL memory controllers using the STOP blocks. This CSR programming is based on the Highland test chip and is not programmable using the firmware. If an alternate configuration is required, programming outside of the firmware must be done. This operation uses an appropriate configuration for both DIV1 and DIV2 memory controllers. A PHY/HBM reset operation must be performed before initializing the memory controllers.

0x20 – Run CTCs – Runs all CTC blocks in parallel to generate traffic to the HBM using the memory controllers. This operation assumes the system has already been reset and the memory controllers have been reset through either firmware or other means. The CTCs for each channel are started in continuous transaction mode generating PRBS data and performing error monitoring. Once all CTCs have been started, the CTCs are stopped after waiting for a delay determined by the hbm_ctc_run_cycles parameter. If any CTCs detect an error, the operation fails. Errors for specific channels can be ignored by setting the hbm_ctc_channel_ignore parameter.

0x21 – Lane Verify – Checks for interposer lane errors. This operation is similar to the 0x0d Lane Repair operation. The only difference is that this operation will fail if any lane errors are detected, even if these are repairable. Note that lane verify ignores any hard lane repairs that may have been previously applied. No soft repairs will be made.

Lane errors are saved to the hbm_spare registers access with interrupt 0x32. These are saved as follows:

hbm_spare[0] – Total number of lanes with faults
hbm_spare[1-23] – Lane number with fault

Each lane number can be mapped to a signal name by referring to the JEDEC HBM Specification's EXTEST_TX WDR definition. The lane repair values match this register definition.

0x22 – Start All CTCs – Starts running all CTCs in continuous mode. This operation will start all CTCs on the Sbus ring running in a continuous mode with error checking enabled. CTC traffic will continue to run until operation 0x23 is called to stop the CTCs. By running this operation on each SBus ring, all CTC s on the chip can be run simultaneously. Since this does not stop the CTCs, traffic can be run for any length of time.

0x23 – Stop All CTCs – Stops all CTCs and checks for errors. Used in conjunction with operation 0x22, this will stop all CTCs that were previously running. If any errors were detected during the CTC run, an error will be returned.

0x24 – Load TMRS Code – Loads a single Samsung TMRS code. This operation will read a single TMRS code set in the APC SPARE registers and load it into the HBM.

SPARE3[31:16] – DIE/SAFETY
SPARE3[15:0] – CATEGORY
SPARE2[11:16] – SUBCATEGORY
SPARE2[15:0] – NAME

Example: Program code CA04 (BDIE=0 SID0/SIDE1=1 SAFETY=1)

SPARE3 = (0b0111 << 16) || 0xa
SPARE2 = (0 << 16) || 0x4

The bypass WDR is set at the end of each TMRS instruction. This is less efficient than setting after all TMRS sequences are programmed but is easier for the user.

0x25 – Release CTC Control – Releases control of the memory controller to the user interface. When running any CTC operations, the AXI port of the memory controller will be enabled and the STOP block will take over control of the APB interface. This operation releases control back to the user interfaces. This will apply to all CTC/STOP blocks on the Sbus chain.

0x26 – CATTRIP Test – Checks HBM CATTRIP functionality. In order to verify CATTRIP operates properly, the CATTRIP MR7 bit is toggled to assert CATTRIP and ensure the PHY received the CATTRIP signal. This verifies that cattrip can toggle but does not check that CATTRIP will trip at high temperatures.

4 Recommended Reset and Initialization Procedure

The following procedure ensures the proper reset for the HBM and PHY at power-on. This procedure is fully documented in the *Broadcom 16 nm HBM PHY Specification*. The following outline includes firmware usage.

1. Power-on the chip and follow all reset recommendations, which includes the following:
   — De-assert SBus Reset
   — De-assert POWER_UP_DRV_DISABLE_L
   — Reset PLL and wait for lock
   — De-assert HBM_POWER_ON_RESET_L
2. Upload firmware.
3. Verify the SBus Master Spico firmware by reading the revision (interrupt 0x0) and build ID (0x1).
4. Program optional firmware parameters (interrupts 0x34 and 0x35). Recommended parameters to set include:
   — hbm_t_rdlat_offset
   — hbm_freq
   — hbm_div_mode - hbm_parity_latency
5. Run the reset operation (interrupt 0x30, operation 0) to reset the PHY. Optionally the power-on flow (interrupt 0x30, operation 5) can be run instead to add basic PHY/HBM verification.
6. Wait for reset/power-on to complete (poll interrupt 0x32, result 0).
7. If errors are detected, read all error codes.
8. The PHY and HBM should now be ready for mission mode operation.

4.1 PHY Configuration and HBM Mode Registers

The HBM PHY contains an IEEE 1500 PHY_CONFIG WDR for programming PHY configuration. This is documented in the *Broadcom 16 nm HBM PHY Specification*. Optimal default values for most of these configuration parameters have been determined based on silicon characterization. The optimal values have been preset in the firmware. During a reset operation, which is included in any operation that runs in test mode, the PHY_CONFIG WDR is automatically programmed to ensure proper operation of the system. The PHY_CONFIG can also be programmed at any time by launching the Reset PHY Config firmware operation. Should the need arise to program alternate values, the PHY_CONFIG default can be changed by setting the following firmware parameters:

hbm_phy_config0
hbm_phy_config1
hbm_phy_config2
hbm_phy_config3
hbm_phy_config4
hbm_phy_config5
hmb_phy_config6

Keep in mind that changing these parameters affects test mode operation as well as mission mode operation.

While most parameters default to the optimal values based on silicon characterization, the T_RDLAT_OFFSET and PARITY_LATENCY parameters (one per DWORD) are commonly changed by the end user. To simplify setting these parameters in the PHY_CONFIG, separate firmware parameters hbm_t_rdlat_offset and hbm_parity_latency are used. The values programmed for these are merged with the hbm_phy_config* parameters when the PHY_CONFIG WDR is programmed.

4.2 HBM Mode Registers

When running any operations either in test mode or mission mode, the HBM mode registers must be programmed. The values required in each mode, however, are not the same. When firmware runs a test mode operation, an HBM/PHY reset is performed and mode registers are programmed using the IEEE 1500 interface with optimal values determined from silicon characterization for running test operations. These are preset in firmware using the following parameters:

hbm_test_mode_register0
hbm_test_mode_register1
hbm_test_mode_register2
hbm_test_mode_register3
hbm_test_mode_register4
hbm_test_mode_register5
hbm_test_mode_register6
hbm_test_mode_register7
hbm_test_mode_register8

When exiting a test mode operation or when the Reset Mode Registers operation is launched, separate mode register values are programmed that are appropriate for mission model. These can be programmed by the end user to any desired value using the following parameters:

hbm_mode_register0
hbm_mode_register1
hbm_mode_register2
hbm_mode_register3
hbm_mode_register4
hbm_mode_register5
hbm_mode_register6
hbm_mode_register7
hbm_mode_register8

While the firmware writes the mode register values upon reset and returning to mission mode, the memory controller may perform its own mode register programming. This ensures that the HBM is programmed consistently with the memory controller expectations.

When modifying the parity latency value in the mode registers, the parity latency in the PHY_CONFIG must also be programmed to match. To ensure consistency, the value of the hbm_parity_latency firmware parameter is used to program both the mode register and PHY_CONFIG parity latency bits automatically, overwriting the values in hbm_mode_register* and hbm_phy_config*.

4.3 CKE

When the PHY enters test mode, it assumes control of all DFI inputs and signals interfacing with the HBM. Upon exiting test mode, control of the interfaces return to normal mission mode operation. This is significant for the CKE signal.

The PHY and firmware control CKE during test mode. When exiting, the CKE state might be different from the mission mode, causing a change to the HBM power-down state. To avoid unexpected power-down state changes, the value of CKE upon exiting test mode is programmable with the hbm_cke_exit_state parameter. Setting this parameter to 1 ensures the CKE is high before exiting test mode. If the DFI CKE signal is also high, the CKE value does not change when transitioning from test mode to mission mode. Setting the hbm_cke_exit_state parameter to 0 ensures the CKE is low before exiting test mode.

5 Direct Control Without Firmware

Performing operations through firmware simplifies access to the PHY and HBM features by providing an automated and consistent interface. All interactions with the HBM and PHY are performed through the SBus. With the correct sequence of commands, the firmware can be bypassed and the features controlled directly by core logic. There are two primary types of functions that firmware performs: reset and IEEE 1500 access. Their basic usage is described in the following section.

5.1 HBM/PHY Reset

Before using the PHY and HBM, a reset must be performed. This reset includes the main HBM reset, the HBM 1500 reset, the PHY FIFO reset, and the PHY 1500 reset. The following SBus sequence is required to perform the full reset.

```
task phy_hbm_reset();
  sbus_driver.write(`APC_ADDR, `APC_INIT_COMPLETE, 32'h0);
  sbus_driver.write(`APC_ADDR, `APC_BIST_EN_REG, 32'hff);
  #200ns

  // Assert the HBM WRSTN reset
  sbus_driver.write(`APC_ADDR, `SBUS_RESET_REG, 32'b0110101);
  #200ns

  // Assert the 1500 PHY WRSTN and the HBM RESET_N
  sbus_driver.write(`APC_ADDR, `SBUS_RESET_REG, 32'b0010001);
  #200us  // tINIT1

  // Deassert the HBM reset, but keep WRSTN_PHY WRSTN_HBM
  // and APC_1500_MACHINE_RST_L asserted
  sbus_driver.write(`APC_ADDR, `SBUS_RESET_REG, 32'b0010011);
  sbus_driver.write(`APC_ADDR, `SBUS_RESET_REG, 32'b0010010);
  sbus_driver.write(`APC_ADDR, `SBUS_RESET_REG, 32'b0010011);
  #500us  // tINIT3

  // Uncomment these two lines to set CKE high
  //sbus_driver.write(`APC_ADDR, `SBUS_RESET_REG, 32'b1010011);
```

```
    //sbus_driver.write(`APC_ADDR, `SBUS_RESET_REG, 32'b1111111);

    // Use the following line to leave CKE low
    sbus_driver.write(`APC_ADDR, `SBUS_RESET_REG, 32'b0111111);
    #200ns // tINIT5

    sbus_driver.write(`APC_ADDR, `APC_BIST_EN_REG, 32'h0);
    sbus_driver.write(`APC_ADDR, `APC_INIT_COMPLETE, 32'hff);
endtask : phy_hbm_reset
```

For Verilog simulations, the tINIT parameters can be reduced to a smaller time as allowed by the HBM simulation model.

After performing HBM and PHY resets, the HBM mode registers and PHY configuration wrapper data registers must be programmed using the IEEE 1500 interface.

5.2 IEEE 1500 Access

The HBM and PHY each contain an IEEE 1500 interface for programming configurations, reading status information, and controlling test features. The following example Verilog code shows one possible implementation for IEEE 1500 access. Included are example tasks to read the temperature and to set the mode registers.

```
// Select HBM / PHY device
`define HBM_SEL 1
`define PHY_SEL 0

// Control 1500 Operations
`define START_WIR_WRITE 4'b0001
`define START_WDR_WRITE 4'b0010
`define START_WDR_READ  4'b0100
`define SHIFT_WRITE_WDR 4'b1010
`define SHIFT_READ_WDR  4'b1100
`define STOP_1500       4'b0000

// Define HBM 1500 register names and lengths
`define HBM_BYPASS                      8’h00
`define HBM_BYPASS_LENGTH               1
`define HBM_DEVICE_ID                   8'h0e
`define HBM_DEVICE_ID_LENGTH            82
`define HBM_TEMPERATURE                 8'h0f
`define HBM_TEMPERATURE_LENGTH          8
`define HBM_MODE_REG_DUMP_SET           8'h10
`define HBM_MODE_REG_DUMP_SET_LENGTH    128

// Define PHY 1500 register names and lengths
`define PHY_BYPASS                      8’h00
`define PHY_BYPASS_LENGTH               1
`define PHY_CONFIG                      8'h11
`define PHY_CONFIG_LENGTH               184

// Global variable used to store the current instruction length
reg [8:0] current_instr_length;

//------------------------------------------
// Sets the current 1500 device, channel and instruction.  The instruction
// length is passed in, but is not used in this task.  Instead it is saved for
// later use on other tasks.
//-----------------------------------------
task wir_write_channel;
  input bit        device_sel;
  input bit [3:0]  channel;
  input bit [7:0]  instruction;
  input bit [8:0]  instr_length;
  reg          error;

  // Save the instruction for later
  current_instr_length = instr_length;

  sbus_driver.write(`APC_ADDR, `WIR_REG, {19'b0, device_sel, channel,
instruction});
  sbus_driver.write(`APC_ADDR, `CONTROL_1500, 'b0001);
  apc_1500_busy_done_handshake(error);
endtask

//-----------------------------------------
// Write the 1500 regsiter data to the APC via sbus and then launches the 1500
// write operation.  The wir_write task must have been called prior to this to
// setup the WIR.  By default the full instruction length is scanned.
// Alternatively, a length and the shift_only bit can be set to perform
// multi-segment scans.
//-----------------------------------------
task wdr_write_data(
  input bit [319:0] data,
  input bit [8:0] length=current_instr_length,
  input bit         shift_only=0
);
  reg          error;

  for (int wr_bits = 0; wr_bits < length; wr_bits = wr_bits+32)
  begin
    case (wr_bits)
      0   :  sbus_driver.write(`APC_ADDR, 8'd4,  data[31:0]    );
      32  :  sbus_driver.write(`APC_ADDR, 8'd5,  data[63:32]   );
      64  :  sbus_driver.write(`APC_ADDR, 8'd6,  data[95:64]   );
      96  :  sbus_driver.write(`APC_ADDR, 8'd7,  data[127:96]  );
      128 :  sbus_driver.write(`APC_ADDR, 8'd8,  data[159:128] );
      160 :  sbus_driver.write(`APC_ADDR, 8'd9,  data[191:160] );
      192 :  sbus_driver.write(`APC_ADDR, 8'd10, data[223:192] );
      224 :  sbus_driver.write(`APC_ADDR, 8'd11, data[255:224] );
      256 :  sbus_driver.write(`APC_ADDR, 8'd12, data[287:256] );
      288 :  sbus_driver.write(`APC_ADDR, 8'd13, data[319:288] );
    endcase
  end

  if (shift_only == 1)
    sbus_driver.write(`APC_ADDR, `CONTROL_1500, {19'b0, length,
`SHIFT_WRITE_WDR});
  else
```

```
    sbus_driver.write(`APC_ADDR, `CONTROL_1500, {19'b0, length,
`START_WDR_WRITE});

  apc_1500_busy_done_handshake(error);
endtask

//------------------------------------------
// Performs a 1500 read operation.  This reads the 1500 registers for all
// selected channel in parallel and stores the results in the APC.  The
// wdr_read_sbus_data must be used to fetch the data from the APC.
//------------------------------------------
task wdr_read(
  input bit [8:0] length=current_instr_length,
  input bit         shift_only=0
);
  reg error;

  sbus_driver.write(`APC_ADDR, `CONTROL_1500, `STOP_1500);

  if (shift_only == 1)
    sbus_driver.write(`APC_ADDR, `CONTROL_1500, {19'b0, length,
`SHIFT_READ_WDR});
  else
    sbus_driver.write(`APC_ADDR, `CONTROL_1500, {19'b0, length,
`START_WDR_READ});

  apc_1500_busy_done_handshake(error);
endtask

//------------------------------------------
// Performs a 1500 read operation.  This reads the 1500 registers for all
// selected channel in parallel and stores the results in the APC.  The
// wdr_read_sbus_data must be used to fetch the data from the APC.
//------------------------------------------
task wdr_read_sbus_data(
  input     [3:0]   channel,
  output    [319:0] result,
  input bit [8:0] length=current_instr_length
);

  reg    [32:0]  data;
  reg    [319:0] mask;

  sbus_driver.write(`APC_ADDR, `READ_CHANNEL, channel);

  result = 0;

  for (int wr_bits = 0; wr_bits < length; wr_bits = wr_bits+32)
  begin
    case (wr_bits)
      0   :  sbus_driver.read(`APC_ADDR, 8'd20, data);
      32  :  sbus_driver.read(`APC_ADDR, 8'd21, data);
      64  :  sbus_driver.read(`APC_ADDR, 8'd22, data);
      96  :  sbus_driver.read(`APC_ADDR, 8'd23, data);
```

```
        128 :  sbus_driver.read(`APC_ADDR, 8'd24, data);
        160 :  sbus_driver.read(`APC_ADDR, 8'd25, data);
        192 :  sbus_driver.read(`APC_ADDR, 8'd26, data);
        224 :  sbus_driver.read(`APC_ADDR, 8'd27, data);
        256 :  sbus_driver.read(`APC_ADDR, 8'd28, data);
        288 :  sbus_driver.read(`APC_ADDR, 8'd29, data);
      endcase
      result = result | (data << wr_bits);
    end

    // Mask out the unused bits since they could have garbage data in them
    mask = (1 << length) -1;
    result = result & mask;

  endtask

  //------------------------------------------
  // After 1500 operations are launched, a handshake must be performed to ensure
  // the commands are properly captured going from the sbus to the 1500 clock
  // domain.
  //------------------------------------------
  task apc_1500_busy_done_handshake;
    output error;
    reg step_error;

    wait_for_1500_done(1, step_error);
    error = error | step_error;

    sbus_driver.write(`APC_ADDR, `CONTROL_1500, `STOP_1500);

    wait_for_1500_done(0, step_error);
    error = error | step_error;
  endtask

  //------------------------------------------
  // Wait for an expected value on the 1500 DONE bit.
  //------------------------------------------
  task wait_for_1500_done;

    reg done;
    reg [31:0] data;
    integer timeout;

    error = 0;
    timeout = 50;
    done = expected ^ 1;

    while ((done != expected) && timeout > 0) begin
      sbus_driver.read(`APC_ADDR, `BUSY_DONE_1500_REG, data);
      done = data[0];
      timeout = timeout - 1;
    end

    if (timeout == 0) begin
      $display( "ERROR: %t Timeout while waiting for 1500 DONE.", $time);
      error = 1;
    end
  endtask

  //------------------------------------------
  // Read the HBM temperature via the 1500 interface
  //------------------------------------------
  task read_1500_temperature;
    reg [320:0] data;
    reg [6:0]   temp;
    reg [3:0]   channel;
    reg         valid;

    // Set the 1500 WIR to read the HBM temperature
    channel = 4'hf;

  // Only read out valid temps
    valid = 1;
    while (valid == 1) begin
     wir_write_channel(
     `HBM_SEL, channel,
     `HBM_TEMPERATURE, `HBM_TEMPERATURE_LENGTH);
      wdr_read();
      wdr_read_sbus_data(channel, data);
      valid = data[7];
      temp = data[6:0];
    end

    $display( "INFO: %t HBM temperature: %0d", $time, temp);

  endtask

  //------------------------------------------
  // Write the HBM Mode registers and reads the results back
  //------------------------------------------
  task write_1500_mode_registers;
    reg [320:0] data;
    reg [3:0]   channel;

    // Set the 1500 WIR to write the mode registers for all channels
    channel = 4'hf;
    wir_write_channel( `HBM_SEL, channel, `HBM_MODE_REG_DUMP_SET,
  `HBM_MODE_REG_DUMP_SET_LENGTH);

    // Perform a 1500 read.
    wdr_read();

    // Write the mode registers
    data[7:0]   = 8'h73;
    data[15:8]  = 8'h10;
    data[23:16] = 8'h25;
    data[31:24] = 8'ha4;
    data[39:32] = 8'h03;
```

```
    data[47:40] = 8'h00;
    data[55:48] = 8'h00;
    data[63:56] = 8'h00;
    wdr_write_data(data);

    // Read back the mode register values from just channel 0
    channel = 0;
    wdr_read();
    wdr_read_sbus_data(channel, data);

    $display( "INFO: %t Channel:0x%0d MR0:0x%0x", $time, channel, data[7:0]);
    $display( "INFO: %t Channel:0x%0d MR1:0x%0x", $time, channel, data[15:8]);
    $display( "INFO: %t Channel:0x%0d MR2:0x%0x", $time, channel, data[23:16]);
    $display( "INFO: %t Channel:0x%0d MR3:0x%0x", $time, channel, data[31:24]);
    $display( "INFO: %t Channel:0x%0d MR4:0x%0x", $time, channel, data[39:32]);
    $display( "INFO: %t Channel:0x%0d MR5:0x%0x", $time, channel, data[47:40]);
    $display( "INFO: %t Channel:0x%0d MR6:0x%0x", $time, channel, data[55:48]);
    $display( "INFO: %t Channel:0x%0d MR7:0x%0x", $time, channel, data[63:56]);

endtask
```

5.3 IEEE 1500 TMRS

For Samsung HBM devices, internal device settings might need to be configured using Samsung's test mode registers (TMRS). These are used for debug and device tuning. TMRS settings can be programmed through the mission mode interface or through IEEE 1500 instructions. This section describes how to use the HBM PHY to program TMRS through the IEEE 1500 interface using Sbus commands. There is a firmware interrupt for programming TMRS values as well.

TMRS commands are provided by Samsung to end users. These commands are in the form of a 4-character code: <die><name><sub_category><category>. Example codes are 'B023', 'C761'. These codes must be translated into TMRS_GEN WDR instructions. Refer to Samsung documentation for a full description of the TMRS_GEN WDR.

TMRS values must be programmed after any MRS programming and while the HBM is in a quiet state, or else the TMRS programming will be lost. After TMRS programming is complete, the WIR must be reset to the bypass instruction.

The following is a Verilog example task for executing TMRS commands using the HBM PHY's IEEE 1500 interface.

```
// Program the Samsung specific TMRS_GEN 1500 WDR.  The BYPASS WIR must be set
when
// TMRS programming is complete.
//
// Example usage:
//   write_tmrs(0, "C750");
//   write_tmrs(0, "C397");
//   write_tmrs(0, "B012");
//   wir_write_channel( `HBM_SEL, 0, `HBM_BYPASS, `HBM_BYPASS_LENGTH);

// WDR definition
`define HBM_TMRS          8'hb0
`define HBM_TMRS_LENGTH   37

// Safety bit (must be cleared after done programming TMRS)
reg tmrs_safety = 0;
```

```
task write_tmrs(
  input bit [3:0] channel,
  input string code
);
  begin
    string die;
    integer category;
    integer subcategory;
    integer name;
    reg [36:0] data;

    // Extract the code and convert to integer
    die              = code[0];
    category         = code[1] - 48;
    subcategory      = code[2] - 48;
    name             = code[3] - 48;

    // Initialize the data
    data = 0;

    // Set the die to receive the instruction
    if(die == "B") begin
      data[36:34] = 3'b100;
      channel = 0;
    end else if(die == "C") begin
      data[36:34] = 3'b011;
    end

    // Set the safety for the first instruction
    if(tmrs_safety == 0) begin
      data[33] = 1;
      tmrs_safety = 1;
    end else begin
      data[33] = 0;
    end

    // Set the category, subcategory and name
    data[32:22] = (1 << name);
    data[21:11] = (1 << subcategory);
    data[10:0]  = (1 << category);

    $display("Setting die         : %04b", data[36:33]);
    $display("Setting name        : %010b", data[32:22]);
    $display("Setting subcategory : %010b", data[21:11]);
    $display("Setting category    : %010b", data[10:0]);

    wir_write_channel( `HBM_SEL, channel, `HBM_TMRS, `HBM_TMRS_LENGTH);
    wdr_write_data(data);
  end
endtask
```

6 Recommended Verilog Test Protocol

To verify connectivity and functionality of the HBM interface, minimal pre-silicon Verilog verification by the customer should include the tests below. Most of these are simple tests that run quickly and can be verified at any design level.

Test	Coverage
1.Upload firmware	SBus connectivity and functionality
2.Query firmware build ID and revision	Spico operation
3.Perform an HBM/PHY firmware-based reset operation	Firmware to HBM/PHY communication and 1500 writes
4.Read HBM Device ID using firmware or direct 1500 access	1500 reads
5.Run the connectivity test operation	Verifies CTC connectivity to each channel
6.Run the power-on flow operation	Verifies AC test operation
7.Run basic CTC/STOP tests (not covered in this document)	CTC/STOP connectivity to memory controller
8.Perform mission mode functional simulations	Memory controller, PHY, and HBM connectivity

7 Revision History

7.1 Version 0.10, April 19, 2018

- Updated Section 1, SBus Master Spico Firmware
- Updated Table 4, HBM Spico Parameters
- Updated Section 2.4, HBM Spico Parameters
- Updated Table 5, Possible Error Codes
- Updated Section 2.5, Error Codes
- Updated Table 7, Operations and Descriptions
- Updated Section 3, HBM Firmware Operations

7.2 Version 0.9. December 1, 2017

- Updated Section 1, SBus Master Spico Firmware
- Updated Section 2.4, HBM Spico Parameters
- Updated Section 2.5, Error Codes
- Updated Section 2.6, Operation Codes
- Updated Section 3, HBM Firmware Operations
- Updated Section 4, Recommended Reset and Initialization Procedure
- Updated Section 5.2, IEEE 1500 Access

7.3 Version 0.8. August 1, 2017

- Updated 0x31 – Launch operation on a single channel and Table 3, Result Types and Descriptions
- Added parameter offsets 0x3e to 0x48 to Section 2.4, HBM Spico Parameters, and Table 4, HBM Spico Parameters
- Updated parameter offset 0x20 description in Section 2.4, HBM Spico Parameters
- Updated the following parameter offset descriptions in Section 3, HBM Firmware Operations:
  - 0x08 – Device ID Test
  - 0x0d – Lane Repair
  - 0x10 – Read Device Temperature
  - 0x13 – Run Samsung MBIST
  - 0x14 – Run SK hynix MBIST
  - 0x21 – Lane Verify
  - 0x22 – Start All CTCs
- Updated Section 4.1, PHY Configuration and HBM Mode Registers
- Updated Section 4.2, HBM Mode Registers
- Updated the code in Section 5.2, IEEE 1500 Access
- Rebranded Avago Technologies to Broadcom.

7.4 Version 0.7, April 18, 2017

- Added new parameters, operations and error codes
- Updated descriptions for reset operation 0x0 to include programming of soft lane repairs.
- Removed operation 0x1a - Load Samsung TMRS
- Corrected example code for programming TMRS values in Section 5.3

7.5 Version 0.6, December 22, 2017

- Added new parameters
- Update hbm_mode_register parameter values
- Added new operations

7.6 Version 0.5, November 9, 2016

- Added new parameters, operations, and error codes
- Added TMRS programming section to Section 3, HBM Firmware Operations
- Added new parameters, operations, and errors codes
- Updated PHY_CONFIG, mode register, and CKE programming description
- Updated Section 4, Recommended Reset and Initialization Procedure

7.7 Version 0.4, August 15, 2016

- Added new parameters, operations, and error codes

7.8 Version 0.3, May 31, 2016

- Corrected code for setting a read channel in wdr_read_sbus_data Verilog task

7.9 Version 0.2, April 22, 2016

- Updated Channel Operation Codes in Table 6, Channel Operation Codes
- Added approximate run times for Lane Repair, AWORD ILB and DWORD ILB tests
- Renamed POWER_ON_RESET_L to HBM_POWER_ON_RESET_L
- Updated phy_hbm_reset Verilog task with latest recommended reset sequence
- Correct instruction length in IEEE 1500 Verilog task examples

7.10 Version 0.1, March 2016

Initial preliminary release.