RADIANT Documentation

P.S. Allison, 2/24/21

This document details the interface to the RADIANT board. The RADIANT has 2 separate communications paths: one is a commanding path which is a simple UART operating at 1 Mbaud. Register reads and writes can be conducted via this path.

The second path is a high-speed SPI path, implemented as a straightforward FIFO. This path essentially acts as a direct memory access (DMA) path for the RADIANT, allowing data to be moved into/out of the address space of the FPGA quickly.

# High-Speed SPI DMA

The high-speed SPI DMA operates at up to 48 MHz as currently implemented, and contains a 2048-entry FIFO on both the receive and transmit paths. The transmit path (*out of* the RADIANT) has a full indicator with a programmable threshold which asserts on FPGPIO0 when full.

A programmable DMA engine allows moving data *to/from* SPI to the internal address space. In addition, data can be read/written either as full 32-bit (little endian) words, or as a single byte to a target. More details are in the SPIDMA Engine register documentation.

# Register Communications Protocol

The RADIANT is commanded primarily via a simple UART packet interface operating at 1 Mbaud (8 bits, no parity, 1 stop bit) with no flow control. From the user’s point of view, both the board manager *and* the FPGA are commanded via the same packets and same interface, with an address space split between the two.

Packets are Consistent Overhead Byte Stuffing (COBS) encoded using 0x00 as a packet delimiter, and consist of a 24-bit address+read/write indicator followed by either number of bytes requested (for reads) or data (for writes).

**Note**: all packets shown here are *before* COBS encoding. Packets are limited to a payload size of 254 bytes. With COBS encoding, this *guarantees* that all packets have a fixed size of payload+2 bytes, while *ensuring* that the packet delimiter (0x00) *always* indicates a packet boundary.

Read Format:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **READ REQUEST** | | | | |
| *Byte 0* | | *Byte 1* | *Byte 2* | *Byte 3* |
| [7] | [6:0] | [7:0] | [7:0] | [7:0] |
| 1 | ADDR[22:0] | ADDR[15:8] | ADDR[7:0] | # of bytes requested – 1 |

Write Format:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **WRITE REQUEST** | | | | |
| *Byte 0* | | *Byte 1* | *Byte 2* | *Byte 3..N (<253)* |
| [7] | [6:0] | [7:0] | [7:0] | [7:0] |
| 0 | ADDR[22:0] | ADDR[15:8] | ADDR[7:0] | DATA[0..N-3] |

In this way up to 250 bytes can be read or written at any given time.

*Typically* the data is processed in incrementing addresses: that is, a 4 byte read from address 0x00 will return address 0x00, 0x01, 0x02, 0x03. *Note*: see the “burst address mode” for clarification here.

## Addressing Notes

The address specified in read/write requests are *byte addresses*, and both the board manager and FPGA all have internal *32-bit little endian* address spaces. This means that *although the addresses are written big-endian* (which is needed to allow quick switching at the board manager level), the data *comes out* ***little-endian***. For example, a read to the IDENT register in the Board Manager returns, in order, 0x4D, 0x42, 0x44, 0x52.

## FPGA Burst Address Mode

Writes/reads to the FPGA are *typically* done with incrementing addresses: an 8-byte write to address 0x20 will write to 0x20-0x27, for instance. However, some operations on the FPGA require repeated writes to the *same address*. In this case, one can switch to *burst address mode*, which requires setting bit 3 in the CONTROL register of the **board manager**. Burst address mode works because for communication between the FPGA and Board Manager, ADDR[22] would always be 0.

Burst address mode consists of 3 separate “sub” modes which are controlled by the reset/mode register in the ***FPGA****.* ***Note the difference:***burst mode *overall* is controlled by a board manager register, whereas burst *addressing type* is controlled by the FPGA. This allows changing from burst<->normal access without any accesses to the FPGA.

* Burst byte address: address never changes (8 byte write to 0x20 writes 8 times to 0x20)
* Burst word address: address increments modulo 2 (8 byte write to 0x20 writes 2 bytes to 0x20/0x21 4 times)
* Burst dword address: address increments modulo 4 (8 bytes write to 0x20 writes 4 bytes to 0x20/0x21/0x22/0x23 2 times)

Switching between modes is detailed in the FPGA register space documentation of the RESET\_MODE register.

# Board Manager Register Space

The RADIANT Board Manager has a 32-bit register space accessed in the same way that the FPGA registers are accessed (via COBS-encoded packets with a 23-bit address + r/w bit, plus a number of bytes requested or data to be written). Board manager registers are accessed as an address space from 0x400000-0x7FFFFF.

Board manager registers should only be accessed at 32-bit offsets (e.g. register 0, 4, 8, 16, etc.), and it’s probably easier to just only do 4 byte reads + writes, although the firmware will not crash or break if non-32 bit values are requested/read.

|  |  |  |
| --- | --- | --- |
| **Address (Base: 0x400000)** | **Name** | **Description** |
| 0x0 | IDENT | “RDBM” (constant returned value: 0x5244424D) |
| 0x4 | DATEVERSION | Version info + date encoding |
| 0x8 | STATUS | Status register (state of various pins) |
| 0xC | CONTROL | Control register (control of some pins) |
| 0x10 | ANAV10 | Analog readback of 1.0V |
| 0x14 | ANAV18 | Analog readback of 1.8V |
| 0x18 | ANAV25 | Analog readback of 2.5V |
| 0x1C | ANALEFT | Analog readback of LEFT monitor (ch 0-11) |
| 0x20 | ANARIGHT | Analog readback of RIGHT monitor (ch 12-23) |
| 0x24 | SPIOUTLSB | Board SPI output (to attenuators) |
| 0x28 | SPIOUTMSB | Board SPI output (to sig generator) |
| 0x30-0x3C | reserved | Reserved |
| 0x40 | GPIO0 | Quad 0 GPIO |
| 0x44 | GPIO1 | Quad 1 GPIO |
| 0x48 | GPIO2 | Quad 2 GPIO |
| 0x4C | GPIO3 | Quad 3 GPIO |
| 0x50 | GPIO4 | Quad 4 GPIO |
| 0x54 | GPIO5 | Quad 5 GPIO |
| 0x58 | SIGGPIO | Signal generator GPIO |
| 0x5C-0x7C | reserved | reserved |
| 0x80 | TDBIAS0 | Trigger diode bias ch 0 |
| 0x84 | TDBIAS1 | Trigger diode bias ch 1 |
| 0x88 | TDBIAS2 | Trigger diode bias ch 2 |
| 0x8C | TDBIAS3 | Trigger diode bias ch 3 |
| 0x90 | TDBIAS4 | Trigger diode bias ch 4 |
| 0x94 | TDBIAS5 | Trigger diode bias ch 5 |
| 0x98 | TDBIAS6 | Trigger diode bias ch 6 |
| 0x9C | TDBIAS7 | Trigger diode bias ch 7 |
| 0xA0 | TDBIAS8 | Trigger diode bias ch 8 |
| 0xA4 | TDBIAS9 | Trigger diode bias ch 9 |
| 0xA8 | TDBIAS10 | Trigger diode bias ch 10 |
| 0xAC | TDBIAS11 | Trigger diode bias ch 11 |
| 0xB0 | TDBIAS12 | Trigger diode bias ch 12 |
| 0xB4 | TDBIAS13 | Trigger diode bias ch 13 |
| 0xB8 | TDBIAS14 | Trigger diode bias ch 14 |
| 0xBC | TDBIAS15 | Trigger diode bias ch 15 |
| 0xC0 | TDBIAS16 | Trigger diode bias ch 16 |
| 0xC4 | TDBIAS17 | Trigger diode bias ch 17 |
| 0xC8 | TDBIAS18 | Trigger diode bias ch 18 |
| 0xCC | TDBIAS19 | Trigger diode bias ch 19 |
| 0xD0 | TDBIAS20 | Trigger diode bias ch 20 |
| 0xD4 | TDBIAS21 | Trigger diode bias ch 21 |
| 0xD8 | TDBIAS22 | Trigger diode bias ch 22 |
| 0xDC | TDBIAS23 | Trigger diode bias ch 23 |
| 0xE0 | VPEDLEFT | Pedestal voltage for ch0-11 |
| 0xE4 | VPEDRIGHT | Pedestal voltage for ch12-23 |

# Register descriptions

## 0x08: STATUS register

The STATUS register covers several status input pins. The bit breakdown is

* Bit 0: FPGA\_DONE (1 if FPGA configuration is complete)
* Bit 1: /MGTDET (0 if an MGT connection is detected – note this is the connector next to the FPGA)
* Bit 2: SD\_DETECT (1 if a micro-SD card is inserted, 0 otherwise)
* Bit 3: PG1V0 (Power Good 1.0V)
* Bit 4: PG1V8 (Power Good 1.8V)
* Bit 5: PG2V5 (Power Good 2.5V)
* Bit 6: PG2V6 (Power Good 2.6V)
* Bit 7: PG3V1 (Power Good 3.1V)

## 0x0C: CONTROL register

The CONTROL register covers several control outputs. These are mostly unused at the moment, but the used bits are:

* Bit 2: /BM\_EN\_10MHz (1 to *disable* the onboard 10 MHz clock)
* Bit 3: BURST bit for FPGA-bound packets (if 1, the BURST bit gets set for outgoing FPGA packets and the address *does not increment* in the same way)

## 0x10-0x20: Analog readbacks

These values are all analog inputs to the Board Manager. They are 16-bit values referenced to 3.3V (e.g. the value is val\*3.3V/65535).

## 0x40-0x54: Quad GPIOs

These are the I2C GPIOs for each quad (group of channels). Quad 0 controls channels 0-3, quad 1 controls channels 4-7, etc.

There are 8 controllable bits in each register. The bits are:

* Bit 0: SEL\_CAL (set to 1 to enable calibration for this quad)
* Bit 1: ATT\_LE (set to 1 to drive LE – latch enable - for attenuators in this quad)
* Bit 2: BIST (set to 1 to enable built-in self-test mode for this quad)
* Bit 3: green/red LED (set to 1 to drive green LED, set to 0 to drive red LED)
* Bit 4: TRIG\_EN (enable 3.0V trigger voltage)
* Bit 5: LAB\_EN (enable 2.6V LAB voltage)
* Bit 6: DIP switch bit 0 (controls TRIG\_EN at startup)
* Bit 7: DIP switch bit 1 (controls LAB\_EN at startup)

Bits 6 and 7 cannot be modified (read-only). All others are read-write.

### GPIO Usage

There are several ways to use the quad GPIOs. Specifically, the BIST bit enables driving the analog monitors from the LABs/quads for use in self-testing. This usage will not be covered here (always keep BIST bit 0).

The SEL\_CAL bit switches all 4 channels in a quad to CALIBRATION mode. Only 1 quad on each side (e.g. one quad of 0, 1, 2 and one quad of 3, 4, 5) can be switched into calibration mode at a time. In calibration mode, either an FPGA-derived impulse or an approximately sine-wave signal is fed to each LAB instead of the RF input (depending on the state of the CAL\_PULSE pins in the SIGGPIO).

The ATT\_LE bit drives the LE (latch enable) for all 8 attenuators attached to this quad. To set a specific attenuator:

* Write control word (A[7:0],D[7:0]) to SPIOUTLSB
* Set ATT\_LE to 1
* Set ATT\_LE to 0

A[7:0] here is the address of the attenuator – only A[2:0] are used. Attenuator addresses are 0, 2, 4, 6 = signal attenuators for quad ch 0, 1, 2, 3 (e.g. channel % 4) and 1, 3, 5, 7 are trigger attenuators for channels 0, 1, 2, 3 (again channel % 4).

**Example**: to set the trigger attenuator for channel 14 (which is the 2nd channel on quad 3, so attenuator address 5) to 50, you would:

* Write 0x532 to SPIOUTLSB. Here 0x5 is the attenuator address, and 0x32 (50) is the setting.
* Read GPIO3 to ‘val’
* Write “val | 0x2” to GPIO3
* Write “val” to GPIO3

**Note:** SPIOUTLSB only uses the low 16 bits of the word written.

## 0x58: SIGGPIO

This is the GPIO connected to the test signal generation section. The bits are

* Bit 0: CAL\_FIL0 (0th bit of test signal filter selection)
* Bit 1: SIG\_LE (Latch enable for the signal generator)
* Bit 2: CAL\_FIL1 (1st bit of test signal filter selection)
* Bit 3: /CAL\_FIL1 (*must be* opposite of bit 2)
* Bit 4: CAL\_PULSE (1 if test signal = pulse, 0 if test signal = sine wave)
* Bit 5: /CAL\_PULSE (*must be* opposite of bit 4)
* Bit 6: SG\_ENABLE (1 if the signal generator is powered on)
* Bit 7: SG\_MUXOUT (read-only MUXOUT bit from signal generator)

The signal generator is an ADF4351, which generates a variable-amplitude, variable frequency output from a reference clock. Programming the ADF4351 is not covered here – Python scripts exist to configure the signal generator.

The ADF4351 is connected to the same SPI bus used for programming the attenuators, but it uses the SPIOUTMSB register. **Note**: SPIOUTMSB uses *all 32 bits* of the word written.

The “CAL\_FIL” bits select which banks are used to filter the sine wave. **Only** use one of the settings below.

|  |  |  |  |
| --- | --- | --- | --- |
| **/CAL\_FIL1** | **CAL\_FIL1** | **CAL\_FIL0** | Frequency Band |
| 1 | 0 | 0 | 50-100 MHz |
| 1 | 0 | 1 | 100-300 MHz |
| 0 | 1 | 0 | 300-600 MHz |
| 0 | 1 | 1 | 600 MHz+ |

## 0x80-0xDC: Trigger DAC outputs

These registers control the trigger diode bias voltage (should be ~1.2V). These registers are 12-bit values ranging from 0-2.0V. That is, a value of 1.2V would be ~2500.

## 0xE0-0xE4: Pedestal DAC outputs

These registers control the LAB4 pedestal outputs. These registers are 12-bit values ranging from 0-3.3V. That is, a value of 1.2V would be ~1500. **Note:** the default pedestal should probably be like, 0.8V or something. I think we used **0.74V** in the LAB4D paper.

# RADIANT FPGA Register Space

The RADIANT FPGA register space begins at a base address of 0. Note that again, all registers are 32-bit. However, the RADIANT can handle individual byte read/write to any address, not just 32-bit aligned addresses. This is helpful for burst streaming to certain registers.

The address space is broken up into 7 separate regions:

|  |  |  |
| --- | --- | --- |
| **Address Region** | **Name** | **Description** |
| 0x000000-0x007FFF | RAD\_ID\_CTRL | RADIANT identification and general control |
| 0x008000-0x00FFFF | SPIDMA | High-speed SPI DMA engine config and control |
| 0x010000-0x01FFFF | LAB4\_CTRL | LAB4 Controller |
| 0x020000-0x02FFFF | LAB4\_RAM | LAB4 RAM |
| 0x030000-0x03FFFF | TRIG | Trigger control and status |
| 0x040000-0x07FFFF | SCAL | Scalers for each trigger input |
| 0x080000-0x0FFFFF | CALRAM | Calibration memory |

Note: this does *not* fill the full register space of the RADIANT (which is 22 bit, up to 0x3FFFFF). The top 2 address bits are currently ignored, but that space should be considered as reserved.

### **RAD\_ID\_CTRL Register Space**

|  |  |  |
| --- | --- | --- |
| **Address (Base: 0x000000)** | **Name** | **Description** |
| 0x0 | IDENT | “RDNT” (constant returned value: 0x52444E54) |
| 0x4 | DATEVERSION | Version info + date encoding |
| 0x8 | CPLDCTRL | CPLD Control Register |
| 0xC | CHANNELDIS | Global channel disable |
| 0x10 | PPSSEL | PPS Selection Register |
| 0x14 | RESET\_MODE | Reset/mode register |
| 0x18 | LED | LED FPGA control |
| 0x1C | JTAGLEFT | JTAG register for left CPLD (channels 0-11) |
| 0x20 | JTAGRIGHT | JTAG register for right CPLD (channels 12-23) |
| 0x24 | SPISS | SPI Flash chip select output |
| 0x28 |  |  |
| 0x2C | DEVICEDNA | Device DNA port |
| 0x30-0x3F | SIMPLESPI | OpenCores SPI controller connected to SPI flash |

### Identification/Version Registers (0x00-0x04)

These registers simply identify the RADIANT (“RDNT”) and the current firmware version. Encoding is the same as the Board Manager.

### CPLDCTRL

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **CPLDCTRL (0x8)** | | | | | |
| Bit[31] | Bit[24] | Bits[23:16] | Bit[15] | Bit[8] | Bits[7:0] |
| RIGHTDONE | Load RIGHTCPLD | RIGHTCPLD[7:0] | LEFTDONE | Load LEFTCPLD | LEFTCPLD[7:0] |

The auxiliary CPLDs on the RADIANT handle fanning out signals to the 24x LAB4Ds. They also multiplex the MONTIMING signals back from the LAB4Ds to a single input to the FPGA. CPLDCTRL allows selecting which LAB4D’s inputs are selected. The MONTIMING input for the right channels (12-23) are selected by RIGHTCPLD[3:0] and the MONTIMING input for the left channels are selected via LEFTCPLD[3:0].

In addition, the CPLDs can also be used for a built-in self-test mode. In this mode, **for a single** LAB4D (again selected by bits [3:0]), the WR[4:0] signals are used to select an analog output *and* the SHOUT pin from that LAB4D is routed to the SSINCR\_TDO pin. BIST mode is enabled by setting RIGHTCPLD[7] and LEFTCPLD[7], and the analog values to be selected are taken from RIGHTCPLD[6:4] and LEFTCPLD[6:4].

Full details on BIST mode are located elsewhere. From a practical standpoint, the CPLDCTRL register is used only to select the MONTIMING input. To select the MONTIMING input for LAB #i, write (0x1000100 | ((i%12) << 16) | (i%12)) to CPLDCTRL. Note that this selects both the given LAB and its diagonally-opposite partner (e.g. channel 0 and 12, channel 11 and 23).

**Bit[31] and Bit[15] are 1 when the CPLDs have been properly programmed.** At initialization, checking these two bits in CPLDCTRL is necessary to ensure that normal operation can occur.

### CHANNELDIS

This is a 24-bit register (defaults to all zeros, so everything enabled) which allows disabling the FPGA-driven outputs. If a bit is 1, the (single-ended) outputs for that LAB4D are disabled.

### PPSSEL

|  |  |  |
| --- | --- | --- |
| **RESET\_MODE (0x14)** | | |
| Bit[31] | Bits[30:8] | Bits[7:0] |
| SELINTPPS | *reserved* | PPSHOLDOFF |

SELINTPPS selects an internally-generated PPS based on the onboard clock if 1. 0 uses the external PPS.

PPSHOLDOFF specifies the number of ~0.65 ms intervals that a new PPS is prevented from being recognized after the first, intended to be used for glitch prevention. It defaults to 10.

### RESET\_MODE

|  |  |  |  |
| --- | --- | --- | --- |
| **RESET\_MODE (0x14)** | | | |
| Bit[31] | Bits[30:29] | Bit[9:8] | Bit[0] |
| JTAGEN | SST\_SELB | BURSTSIZE | MMCMRESET |

JTAGEN must be 0 for normal operation. JTAGEN=1 is used for initial CPLD programming.

SST\_SELB must be 0 for normal operation. Setting bit [1:0] to 1 stops the left/right LAB4D clocks (SST).

BURSTSIZE determines how incoming commands are parsed in BURST mode (bit 23 set, controlled by the CONTROL register in the Board Manager. If BURSTSIZE=0, incoming data is handled byte-by-byte. If BURSTSIZE=1, incoming data is handled 2 bytes at a time. If BURSTSIZE=2, incoming data is handled 4 bytes at a time. BURSTIZE=3 is reserved. As an example, if the incoming data is 0x01, 0x02, 0x03, 0x04, the address is 0 and BURSTSIZE=0, this will be interpreted as 4 single-byte writes to address 0: 0x01, 0x02, 0x03, 0x04. If BURSTSIZE=1, this will be interpreted as 2 word writes to address 0: 0x0201 and 0x0403. If BURSTSIZE=2, this is a single 32-bit write to address 0: 0x04030201.

MMCMRESET resets the entire FPGA LAB4D clocking infrastructure, and should not be needed.

### LED

To be documented

### JTAGLEFT/JTAGRIGHT

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **JTAGLEFT (0x1C) and JTAGRIGHT (0x20)** | | | | | | |
| Bit[31] | Bits[30] | Bit[29] | Bits[26:24] | Bits[23:16] | Bits[15:8] | Bits[7:0] |
| Busy | Enable Sequence | Reverse TDO Bitorder | Number of Bits to Clock | TDO Return Values | TMS Output Values | TDI Output Values |

These registers control the JTAG sequence for programming the CPLDs. The programming sequence is not covered here: a Python implementation of the programming sequence exists at <https://github.com/RNO-G/radiant-python> in the RadCPLD module, and an example programmer exists at examples/radcpldprog.py.

### SPISS (0x24)

SPISS is used to talk to the SPI Flash module. Bit 0 drives the chip select for the SPI flash. All other bits are unused.

### DEVICEDNA (0x2C)

This allows extracting the serial number of the FPGA. Write 0x80000000 to begin the readout process, then read this register 57 times to extract the 57 bits of the device DNA using bit 0.

### SIMPLESPI (0x30-0x3F)

This address space contains the OpenCores SPI core for talking to the SPI flash to reprogram the FPGA. Details are not covered here: the overall method for talking to the flash is covered in the spi.py module in radiant-python.

## **SPIDMA Register Space**

|  |  |  |
| --- | --- | --- |
| **Address (Base: 0x008000)** | **Name** | **Description** |
| 0x0 | CONFIG | DMA configuration register |
| 0x4 | CONTROL | Resets, DMA initiation |
| 0x8 | CURDESCR | Current descriptor being processed |
| 0xC | TXNCOUNT | Number of DMA transactions done. Write to reset. |
| 0x80-0xFF | DESCRIPTORS | DMA descriptors 0-31 (32-bits each) |

### CONFIG (Offset 0x00: 0x8000)

The CONFIG register sets up the behavior of the DMA engine. These properties are used for the entire DMA process.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **SPIDMA CONFIG (Offset 0x00: 0x8000)** | | | | | | | | | | | |
| [31] | [30] | [26:16] | [15:9] | [8] | [7:6] | [5] | [4] | [3] | [2] | [1] | [0] |
| TX Full Flag Enable | TX Full Flag Value | TX Full Flag Threshold | Cycle Delay | Enable SPI Receive | Byte Target in Byte Mode | Byte Mode | Reserved | DMA Direction (1=from SPI) | Ext DMA Req Enable | DMA Busy | DMA Enable |

Note that enabling DMA does *not* **start** DMA. To start the DMA, a request must be received – either external (if enabled – this will come from the trigger core) or a soft request by writing to the CONTROL register.

Cycle Delay increases the number of cycles between each DMA transfer. By design a cycle delay of 0 still results in a ~20 clock delay currently. A DMA sequence to certain registers may require more time than this, therefore CYCLEDELAY allows that to be increased up to an additional 127 clock cycles.

Byte Mode works similarly to BURSTSIZE=0. In this case, only a single byte will be written to/read from SPI from the given address. The byte to be selected is specified by bits [7:6] (if 0=bits[7:0], if 1=bits[15:8], etc.). If Byte Mode = 0, the Byte Target field is ignored.

As an example, if a DMA is initiated with Increment Byte Mode=0, and DMA Direction=0, and the descriptor is address=0 and length=1 (address increment doesn’t matter here, obviously), the DMA engine will write [84, 78, 68, 82] to the High Speed SPI interface, corresponding to 0x54, 0x4E, 0x44, 0x52 in the IDENT register. If Byte Mode=1 and Byte Target=0, if address = 0 and length =4 and address increment = 0, the DMA will write [84, 84, 84, 84] to the High Speed SPI interface – the same value from consecutive reads of byte 0, address 0.

Any address from 0x000000-0x0FFFFF can be targeted by the DMA engine.

**Note**: Cycle Delay is necessary to use the DMA engine for certain registers, like the JTAG or Simple SPI cores, as those operations require some time between accesses and DMA can occur very fast. For internal memory (calibration RAM, LAB4 RAM, etc.) it can always be set to 0.

### CONTROL (Offset 0x04: 0x8004)

|  |  |  |  |
| --- | --- | --- | --- |
| **SPIDMA CONTROL (Offset 0x04: 0x8004)** | | | |
| Bit[3] | Bit[2] | Bit[1] | Bit[2] |
| DMAREQ | ENGINERESET | RXRESET | TXRESET |

Each of these bits is self-clearing.

A write to DMAREQ initiates a DMA transfer. This will *always* happen regardless of whether or not the external DMA request is enabled, and if the external DMA request is disabled, this is the only way to initiate DMA.

ENGINERESET halts the DMA engine, clearing DMA Enable (bit 0 in SPIDMA CONFIG). Improper configuration of the DMA engine can result in infinite operation – if this occurs, an ENGINERESET will halt the DMA.

RXRESET resets the receive path and eliminates all data in the RX FIFO.

TXRESET resets the transmit path and eliminates all data in the TX FIFO.

### CURDESCR (Offset 0x08: 0x8008)

This register outputs the current descriptor being processed if a DMA is active.

### TXNCOUNT (Offset 0x0C: 0x800C)

This register outputs the number of DMA transactions that have been processed. Write anything to reset to 0.

### DMA Descriptors (Offsets 0x80-0xFF: 0x8080-0x80FF)

|  |  |  |  |
| --- | --- | --- | --- |
| **SPIDMA Descriptors (Offsets 0x80-0xFF: 0x8080-0x80FF)** | | | |
| Bit[31] | Bit[30:19] | Bit[18] | Bit[17:0] |
| LASTDESCR | CYCLECOUNT | INCREMENT | ADDRESS[19:2] |

LASTDESCR **must** be set in one of the descriptors for DMA to terminate!

CYCLECOUNT is the number of desired reads/writes for this descriptor, minus 1 (so 0 means 1 read/write).

If INCREMENT=1, this descriptor increments (by 4!) the bus address on each cycle.

ADDRESS is the source/destination address for this descriptor, shifted down by 2 (making it a 32-bit address). This gives a range of 0x00000-0xFFFFF, which corresponds to the complete range currently implemented (22 bits total, less 2 at the top and 2 at the bottom).

There are 32 available descriptors: DMA transactions always start with descriptor 0 and increment.

The DMA descriptor method allows for creating arbitrary SPI DMA transactions: for instance, the event FIFO, then entire LAB4 RAM, followed by the scaler space can be read by setting descriptors 0 to the event FIFO, 1-24 for the LAB4 RAM output, and then descriptor 25 for the scalers. Combined with the external DMA request, this turns the SPI output into a direct event streaming output.

## **LAB4\_CTRL Register Space**

In progress.7

## **LAB4\_RAM Register Space**

|  |  |  |
| --- | --- | --- |
| **Address (Base: 0x020000)** | **Name** | **Description** |
| 0x0000-0x07FF | FIFO\_CH0 | Channel 0 LAB Data FIFO |
| 0x0800-0x0FFF | FIFO\_CH1 | Channel 1 LAB Data FIFO |
| … | … | … |
| 0xB800-0xBFFF | FIFO\_CH23 | Channel 23 LAB Data FIFO |

The LAB4 RAM space contains the data *after* passing through the CALRAM. Note that this space is *actually* a FIFO space – reading from *any* register in the corresponding LAB data FIFO will advance the FIFO. So repeated reads from the same address will get new data.

Also note that the address space is *not large enough* to treat as a RAM for a full event.

Therefore, when setting up DMA descriptors for event DMA, *do not* use the address increment option. Read from the same address as many times as needed. Therefore reading all 24 channels will require 24 descriptors.

The FIFOs are reset from the EVENTCTRL register in the TRIG space.

### Data Format

Note that data from the LAB4Ds is naturally grouped in windows of 128 samples. There are *currently* 8 of these windows in a single event. Windows are *always read out in the same order.* The data from the LAB4Ds is a 12-bit value, padded to 16 bits with a 4-bit header:

|  |  |  |  |
| --- | --- | --- | --- |
| **LAB4D Data (any address in LAB4 RAM space)** | | | |
| Bit[15:14] | Bit[13] | Bit[12] | Bit[11:0] |
| BANK[1:0] | STOP | *Reserved* | LAB4VAL[11:0] |

BANK lists which one of the 4 LAB4Ds this group of windows comes from. STOP indicates that this window is the last window of the readout (implying that the *next* window following this is the *beginning*). In other words, if we label the windows as 0-7, if the STOP bit is set on window 6, the *actual* data order should be considered as window 7, 0, 1, 2, 3, 4, 5, 6.

## **TRIG Register Space**

|  |  |  |
| --- | --- | --- |
| **Address (Base: 0x030000)** | **Name** | **Description** |
| 0x0000 | EVENTCTRL | Event control register |
| 0x0004 | PPSCNT | Current PPS count |
| 0x0008 | SYSCLKCNT | SYSCLK count at last PPS |
| 0x000C | LASTCLKCNT | SYSCLK count at previous PPS to the last one |
| 0x0100 | **EVIDENT** | Reads back as “RDE0”. Identifies start of event. |
| 0x0104 | **EVSECOND** | PPS count of event |
| 0x0108 | **EVCOUNT** | Event count of this event |
| 0x010C | **EVSYSCLK** | SYSCLK count of event |
| 0x0110 | **EVINFO** | 32-bit information corresponding to trigger (to be determined) |
| 0x0114 | **EVSTATUS** | Status flags for event. |
| 0x0118 | **EVSYSCLKCNT** | SYSCLK count at PPS before this event |
| 0x011C | **EVLASTCLKCNT** | SYSCLK count at PPS previous to one before this event |
| 0x0200-0x03FF | TRIGTHRESH | Thresholds for the onboard trigger |

Bolded registers are actually FIFOs – reading from them automatically progresses to the next event value.

### EVENTCTRL(Offset 0x00: 0x30000)

|  |  |  |
| --- | --- | --- |
| **EVENTCTRL (Offset 0x00: 0x30000)** | | |
| Bit[21:16] | Bit[1] | Bit[2] |
| PENDING[5:0] | SYNC | FIFORESET |

Writing 1 to FIFORESET resets the event FIFOs (the registers in 0x0100-0x011C, **plus** the LAB4D RAM registers).

Writing 1 to SYNC resets all counters at the *next* PPS. To sync multiple boards, pick one board, read the PPSCNT register until the value changes, and then write 1 to the SYNC register on all boards. This will synchronize all boards. Note that if the PPS is synchronous to SYSCLK, the sync will be perfect, otherwise there may be a +/- slight uncertainty at each SYNC – however if the same SYSCLK is used on all boards this should be obvious because the resulting SYSCLKCNT values will be shifted, on average.

PENDING contains the number of pending DMA requests.

### PPSCNT (Offset 0x0004: 0x30004)

Number of PPS pulses seen since last SYNC.

### SYSCLKCNT (Offset 0x0008: 0x30008)

Value of a free-running SYSCLK counter captured at the last PPS.

### LASTCLKCNT (Offset 0x000C: 0x3000C)

Value of free-running SYSCLK counter captured at PPS *prior* to last PPS. Difference between SYSCLKCNT and LASTCLKCNT (modulo rollover) gives the SYSCLK frequency.

## **SCAL Register Space**

In progress.

## **CALRAM Register Space**

|  |  |  |
| --- | --- | --- |
| **Address (Base: 0x080000)** | **Name** | **Description** |
| 0x0-0x3FFF | CAL\_CH0 | Channel 0 Calibration Data |
| 0x4000-0x7FFF | CAL\_CH1 | Channel 1 Calibration Data |
| … | … | … |
| 0x5bfff-0x5ffff | CAL\_CH23 | Channel 23 Calibration Data |
| 0x60000 | CONTROL | CALRAM control register |
| 0x60004 | MODE | CALRAM mode register |
| 0x60008 | ROLLCOUNT | CALRAM roll count register |

### CALRAM Functionality

The CalRam is designed to generate and store calibration data for the LAB4D. It consists of a pair of digital signal processors (math units) hooked up to local FPGA memory. It can:

* Generate pedestal data for the LAB4Ds by summing inputs in the local memory
* Generate timing data for the LAB4Ds by counting the number of times a sine wave crosses zero, and storing that data in the local memory
* Adjusting incoming data from the LAB4Ds by a value inside the local memory to pre-compensate the pedestal

***Important Note 1:*** CALRAM control/mode registers should *always* be written together (control, then mode) and must be initialized. The “power-on default” of the CALRAM results in *no output data!*

***Important Note 2:*** CALRAM calibration routines (pedestal generation, zero-crossing generation) should be done from a ***fresh start*** of the LAB4 controller, which ensures that the LAB4 samples begin with address 0. Before any calibration run, the LAB4 controller should be stopped, the CalRams zeroed, then the LAB4 controller started again.

### Calibration Primer

The LAB4D sampling chip needs some calibration data collected. Rather than *fully* take data and storing it in the SPI interface (which, while faster than UART, is still slower than the LAB4Ds can generate data), the RADIANT FPGA can calculate that calibration data *on-chip*, allowing data to be collected at *much* higher rates. There are 2 kinds of calibration data: pedestal data, and zero-crossing data.

#### Pedestal Data

Each of the 4096 samples on the LAB4D has an individual “pedestal” which varies slightly from sample to sample. To calculate this pedestal, a reasonable amount of “zero-input” data should be taken. Note that the *easiest* way to do this is to max out the signal attenuators on each input and take data for all channels at once. While it *would* be possible to use the calibration signal switch for this (SELxx\_CAL), doing so would turn on all of the calibration amplifiers, resulting in a large current draw. Maxing out the signal attenuators should reduce the input signal to a small enough value that an average of a few hundred samples gets a well-measured pedestal.

#### Zero-Crossing Data

The LAB4D internal timing is determined from a 128-sample delay chain. The individual delays of those channels typically vary without calibration by ~5%. That sample-to-sample variation may be “trimmed” out using internal calibration DACs. In addition, however, the alignment of the full 128-sample delay chain to the clock *must* be calibrated using the “VTrimFB” register. This “trim” DAC sets the “average” speed for the delay chain.

Calculating times for each individual sample can be done in many ways, but the *simplest* way which has been commonly used is to put in a sine wave, and randomly sample how often a sample occurs that crosses from negative to positive in a set of trials. We call this the “zero crossing count.”

For each of the delay entries in the chain (from 1-127), the number of zero-crossings for that cell is calculated, and with the number of trials, the time can be calculated as: t = (zero-crossing fraction)/frequency.

Note that because the full array of the LAB4D is 4096 samples, but the delay chain is only 128 samples, the *full* zero-crossing data is calculated by summing samples modulo 128 (e.g. add 1, 129, 257, etc.) *However*, the “seam” samples (0, 128, 256, etc.) are special – only 3/8 of them are valid. To get the seam sample, add 256, 512, and 768.

## Channel Calibration Data

Channel calibration data is calculated and stored in a 4096-entry 27-bit RAM for each channel, corresponding to each sample in the LAB4D.

In pedestal mode, the value in the RAM is the sum of all samples fed into the CALRAM. In zerocrossing mode, bits[8:0] are the number of times that this sample was positive (above pedestal) and the previous sample was negative or at pedestal. In zerocrossing mode, bits[26:9] are the (integer) pedestal value.

**Note**: since in zerocrossing mode bits[26:9] are uninteresting (they’re unchanged) the CALRAM MODE register has a ZC\_READ\_MODE bit (bit[2]) which forces the outputs of those bits to be zero, so that reads *only* return the zerocrossing counts. This bit does *not* affect writes. Most of the time when in zerocrossing mode, ZC\_READ\_MODE should be set.

Calibration data for channel 0 sample 0 is at address 0x0, sample 1 is at address 0x4, and then channel 1 sample 0 starts at 0x4000, etc.

### CONTROL (Offset 0x60000: 0xE0000)

|  |  |  |  |
| --- | --- | --- | --- |
| **CALRAM CONTROL (Offsets 0x60000: 0xE0000)** | | | |
| Bit[3] | Bit[2] | Bit[1] | Bit[0] |
| ROLLCOMPLETE | ZCFULL | RESETCOUNTER | ENABLE |

ENABLE must be set for calibration *generation*. When in this mode, data *does not propagate* to the output FIFOs and events are not generated. When ENABLE is 0, data propagates to the output FIFOs.

RESETCOUNTER resets the roll counter. This should be done before a zero-crossing run to ensure that ROLLCOUNT is accurate.

ZCFULL becomes set when the CALRAM was disabled due to pending overflow in zero crossing mode. Because the counting in ZC mode is random, if one of the samples reaches its maximum value (511), at the end of that 4096-sample run, the CALRAM will disable itself to prevent overflow, and future data will be ignored.

ROLLCOMPLETE is set when a full set of 4096 samples has been acquired (4 readouts of 1024 samples each, currently). ROLLCOMPLETE clears as soon as *more* data comes in.

### MODE (Offset 0x60004: 0xE0004)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CALRAM MODE (Offsets 0x60004: 0xE0004)** | | | | |
| Bit[4] | Bit[3] | Bit[2] | Bit[1] | Bit[0] |
| ADJUST\_SIGN | ADJUST\_MODE | ZC\_READ\_MODE | ZERO\_INPUTS | ZC\_MODE |

ZC\_MODE determines if the CALRAM is in pedestal mode (0) or zero-crossing mode (1).

ZERO\_INPUTS is used to quick-zero the CALRAM. Set ZERO\_INPUTS=1, and run a full roll (4096 samples) and all *writeable* CALRAM for that mode will be zeroed. If ZERO\_INPUTS=1 and ZC\_MODE=1, *only* the low 9 bits will be cleared. Note: this means that if a pedestal run of 512 rolls is performed, the pedestals will already be in the correct format for ZC mode and with ZERO\_INPUTS=1 and ZC\_MODE=1, the zerocrossing RAMs can easily be erased.

ZC\_READ\_MODE is used to ensure that the *readout* of the CALRAMs (from the bus, not internal to the CALRAM itself!) only contains the zero-crossing data (the low 9 bits), and not the pedestal. This bit does NOT affect writes!

ADJUST\_MODE, if set, means that the *output data* (to the FIFOs) is *adjusted* by the low 9 bits of the CALRAM for that sample. ADJUST\_SIGN determines whether that value is *added* (ADJUST\_SIGN=0) or *subtracted* (ADJUST\_SIGN=1). Generally ADJUST\_SIGN should be *negative* if ADJUST\_MODE is desired, since the low dynamic range of the LAB4D is unused (values below ~400 or so are not possible).

Note that the output to the FIFOs is *purely* 12 bits: if in ADJUST\_MODE the value overflows/underflows it simply rolls over (e.g. if 4090 is adjusted upward by 100, the output will be 94).

LAB4D Startup Procedure

This section covers the LAB4D startup procedure. This does *not* cover the initial RADIANT setup – therefore, we assume the following state:

* FPGA is programmed
* LAB4D clock is running
* Both CPLDs are programmed and are not in JTAG mode
* All LAB4D power supplies are on (main 2.6V and individual 2.5V supplies)
* DMA is reset and ready

# Get to Configuration Mode and Reset All Registers

1. Ensure the LAB4 controller is not in run mode by checking bit 2 (0x4) in LAB4\_CTRL CONTROL register (0x10000). If bit 2 is set, read the value in the LAB4\_CTRL CONTROL register, clear bit 1, and update that register. Wait until bit 2 is now clear. (Bit 1 *requests* the mode change, Bit 2 indicates the *current mode*).
2. Issue a REGCLR by setting bit 16 in LAB4\_CTRL CONTROL (0x10000), and then clearing it.

# Load All Register Values

LAB4 registers can be loaded simultaneously to *all* devices by writing to a magic LAB4D address of 31 (0x1F). The general registers are common between all LAB4Ds: only the timing registers vary. In the Python core, the register load procedure is in the LAB4\_Controller “default” function.

# Synchronize LAB4D Phases

Internally, the LAB4D toggles between 2 phases, indicated by a PHAB signal. Those phases must be identical between all chips *and* must be matched to the internal phase inside the FPGA so the proper write address is issued. This is done using the LAB4D phase scanner.

# Disable Testpattern Mode

The LAB4 controller comes up outputting only test pattern data from the LAB4D to ensure that *some* output is possible straight away (if none of the registers are programmed, the actual *output* data from the LAB4 will be zero). Therefore for proper data taking, testpattern mode must be disabled.

# Set CALRAM Mode

By default the CALRAM will output *no data* – therefore, the CALRAM CONTROL and MODE registers must be programmed to the desired value before operation.

# LAB4D Event Running Procedure

1. Stop the LAB4 controller.
2. Reset the FIFOs via the EVENTCTRL register’s FIFORESET.
3. Reset the DMA engine and the TX path.
4. Program the DMA engine descriptors. Typically this would be the event header (0x030100, length 8, incrementing address) followed by the LAB4D space (0x020000, length 512, no incrementing address, then 0x020800, length 512, no incrementing address, etc. until the final LAB space, with that descriptor being a terminator)
5. Enable the DMA engine to respond to external requests and use the TX path. Enable the programmable full output if desired.
6. Start the LAB4 controller.

Determining when an event is ready can then be done by waiting for the prog\_full GPIO output to go high. This is FP\_GPIO[0]. This can also be done by polling for bit[30] to go high in the SPIDMA CONFIG register.

Once the data is ready, a read of the full amount (in this case, 49,184 bytes) should succeed.