# POS-PHY Level 4 MegaCore Function v2.1.0 Wrapper Features

## Introduction

The Altera® POS-PHY Level 4 MegaCore® function provides high-speed cell and packet transfers between physical (PHY) and link layer devices. The packet over SONET/SDH physical layer (POS-PHY) Level 4 interface, first developed by the SATURN® Development Group, was later adopted by the Optical Internetworking Forum (OIF) as the System Packet Interface Level 4—Phase 2 (SPI-4.2). Therefore, POS-PHY Level 4 and SPI-4.2 are synonymous. This application note uses SPI-4.2 to refer to the protocol or interface, and uses POS-PHY Level 4 to refer to the MegaCore function.

The SPI-4.2 interface supports a data width of 16 bits (LVDS solution), and can be a PHY-link, link-link, link-PHY, or PHY-PHY connection in multi-gigabit applications, including: asynchronous transfer mode (ATM), packet over SONET/SDH (POS) (STS-192/STM-64), 10 Gigabit Ethernet, and multi-channel Gigabit Ethernet.

The POS-PHY Level 4 MegaCore function is provided via a MegaWizard® Plug-In. This wizard guides designers through parameter selection, and delivers the core with the chosen parameters. The wide range of parameters that the wizard provides is usually all that designers, such as yourself, ever need or want. Sometimes, however, you may be looking for some functionality—offered by the core—that is beyond what the wizard delivers. This application note explains the extra features available in the core and how to use them.

### Wrapping the Core

When the POS-PHY Level 4 MegaCore function is delivered, it consists of more than just an encrypted netlist. Each core delivered includes a pair of files that are used to instantiate a core with the correct feature set. This pair of files, known as wrapper files, consist of a top-level wrapper file that is created by the wizard when you click the **Finish** button, and a core wrapper file that is delivered with the encrypted netlist and is used to tie the core's components together. shows a typical design with a POS-PHY Level 4 MegaCore function. The names of the receiver and transmitter top-level wrappers used in this example are: **my_rx_core** and **my_tx_core**. The top-level filenames and module names for your design would match the names you chose for the cores when you created them in the wizard. The top-level wrappers each instantiate a core

wrapper, with a filename like: **posphy4_tx183_mw_wrapper.v**, where the tx183 is the configuration number. See Figure 1. The core wrappers instantiate the components that make up the core.

*Figure 1. Typical Design Hierarchy*

```
my_design
    my_rx_core (my_rx_core.v)
        pl4_rx183 (posphy4_rx183_mw_wrapper.v)
            aot1139_rx10_atltop
            aot1139_rx183_hssi_rx
            aot1139_rx183_posphy4
            aot1139_rxstat_10_256_rxstatproc

    my_tx_core (my_tx_core.v)
        pl4_tx183 (posphy4_tx183_mw_wrapper.v)
            aot1139_tx10_atltop
            aot1139_tx183_posphy4
            aot1139_tx183_altlvds_tx
```

**Transmitter Cores**

Figure 2 on page 3 shows the relationship between the top-level wrapper, the core wrapper, and the components that make up the MegaCore function, for a typical transmitter. Instantiate the top-level wrapper in your SPI-4.2 design.

*Figure 2. Transmitter Wrapper Structure*
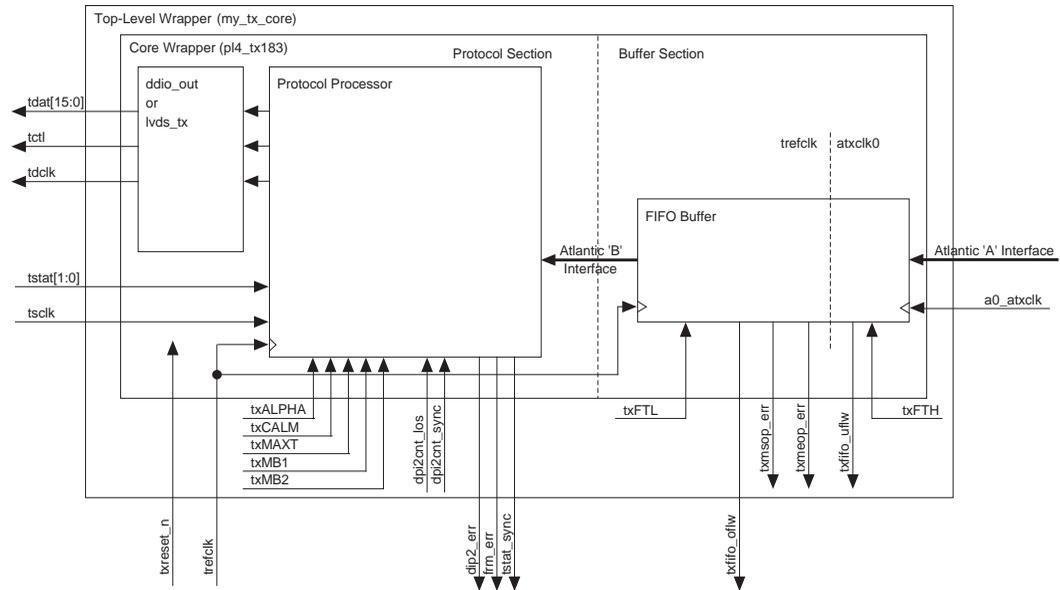


## Transmitter Top-Level Wrapper Structure

The top-level wrapper is exceedingly simple. It does nothing other than to instantiate the core wrapper. All of the ports in and out of the core wrapper are passed through to be used by the application logic, with the exception of the parameters set by the wizard. The wizard writes these parameters (see Table 1 on page 4) into the top-level wrapper, based on the values you selected in the graphical user interface (GUI) to create the core. When writing these values, the wizard translates from units of bytes used in the GUI to units of FIFO elements used by the core, where FIFO elements are particular to the core type. (See Table 2 on page 5 for a summary). Each parameter in the top-level wrapper has a corresponding signal named `signal_wiren`, where *n* is different for each parameter. By editing the wrapper file, you can change the values of these input signals to change the core parameters, or you can connect these signals to registers to allow software to change the core parameters during operation.

☞ Keep in mind that the units in the wrapper file are not the same as the units in the MegaWizard Plug-In.

| Table 1. Transmitter Top-Level Wrapper Parameters  (Part 1 of 2) | | |
|---|---|---|
| **Name** | **Signal Name** *(1)* | **Description** |
| txALPHA | signal_wire0[7:0] | The number of times to repeat the training pattern when training is inserted. When controlled by software, changes to this parameter take effect at the beginning of the next training pattern transmission. **Range**: $0 \leq$ txALPHA $\leq$ 0xFF, 0 = disable **Units**: Number of repetitions **Clock**: trefclk |
| txMAXT | signal_wire1[15:0] | The time between training pattern insertions. When controlled by software, changes to this parameter take effect after the transmission of the next training pattern. **Range**: $2 \leq$ txMAXT $\leq$ 0xFFFF **Units**: 8 SPI-4.2 cycles (16 bytes of data or control words) **Clock**: trefclk |
| txFTH | signal_wire2[n:0] | The threshold (measured from the top of the FIFO buffer) used to indicate that the buffer is getting full. Above this threshold, backpressure is asserted to the application transmit logic to indicate that no more data should be pushed into the transmit buffer. Changes to this parameter take effect immediately. *n* depends on the buffer depth. **Range**: $0 \leq$ txFTH < FIFO size **Units**: FIFO elements **Clock**: Atlantic clock |
| txFTL | signal_wire3[n:0] | The threshold (measured from the bottom of the FIFO buffer) used to indicate that there is enough data in the buffer to warrant that the transmitter start sending data. Changes to this parameter take effect immediately. This should be set to a value of at least 16 bytes to maintain transfers that are multiples of 16 bytes as required by the SPI-4.2 specification. *n* depends on the buffer depth. **Range**: $0 \leq$ txFTL < FIFO size **Units**: FIFO elements **Clock**: trefclk |
| txMB1 | signal_wire4[6:0] | The burst size to send when a starving status is received. Changes to this parameter take effect at the beginning of the next burst. **Range**: txMB2 $\leq$ txMB1 $\leq$ 0x7F **Units**: 8 SPI-4.2 data cycles (16 bytes of data) **Clock**: trefclk |
| txMB2 | signal_wire5[6:0] | The burst size to send when a hungry status is received. Changes to this parameter take effect at the beginning of the next burst. **Range**: $8 \leq$ txMB2 $\leq$ txMB1 **Units**: 8 SPI-4.2 data cycles (16 bytes of data) **Clock**: trefclk |

**Table 1. Transmitter Top-Level Wrapper Parameters  (Part 2 of 2)**

| Name | Signal Name *(1)* | Description |
|------|-----------|-------------|
| `txCALM` | `signal_wire6[7:0]` | The number of times to repeat the status message before the DIP-2 code is sent. Changes to this parameter take effect immediately.<br>☞      This setting must match the receiver's `CALM` setting.<br><br>**Range**: 1≤ `txCALM` ≤ 0xFF<br>**Units**: Number of repetitions<br>**Clock**: `trefclk` |

*Note to Table 1:*
(1)   The signal names are generated by the MegaWizard Plug-In.

**Table 2. Transmitter FIFO Element Sizes**

| Transmitter Data Path Width (bits) | Atlantic Data Width (bits) | Transmitter FIFO Element Size (bytes) |
|---|---|---|
| 32 | 32 | 8 |
| 64 | 64 | 16 |
|  | 128 |  |
| 128 | 128 | 32 |
|  | 256 |  |

## Transmitter Core Wrapper Structure

The core wrapper is more complex than the top-level wrapper; it ties together the various blocks that make up the POS-PHY Level 4 MegaCore function. In addition, it includes extra features that can be accessed by editing the source file. This section describes the structure of the core, and how the components are wired together.

Figure 2 on page 3 shows how the core wrapper is divided into two sections: a protocol section and a buffer section. The protocol section includes the protocol processor, and the LVDS or double data rate input/output (DDRIO) component. The buffer section includes the FIFO buffer(s) and, depending on the configuration, multiplexers to feed data from the buffers into the protocol processor. The details of the buffer section change drastically depending on the configuration.

### Protocol Section

The protocol section is responsible for handling everything related to the SPI-4.2 protocol.

**Protocol Core (ipcore)**

The protocol core receives and handles the status message from the receiver, schedules data to be transmitted, and formats the data so that it conforms to the SPI-4.2 protocol. The protocol core instantiation within the core wrapper Verilog HDL file is similar to the following code snippet. If you are using VHDL, the instance and component names are still the same.

```
aot1139_tx230_posphy4 ipcore( <signal list> );
```

The *aot1139* name identifies the build of this MegaCore function; *tx230* is the configuration number. The configuration number may be different for cores with different wizard parameters.

**LVDS Output Block (altlvds_tx)**

For 64- and 128-bit transmitter cores, an altlvds_tx component is instantiated to allow high-speed operation. This component is a hard macro in the device's I/O block that performs serialization and clock multiplication. The instantiation within the core wrapper file is similar to the following code snippet:

```
aot1139_tx10_altlvds_tx txlvds ( <signal list> );
```

The aot1139_tx10_altlvds_tx module is defined within the same file as the core wrapper, and simply instantiates Altera's altlvds_tx component with the appropriate parameters. For details on the altlvds_tx component, refer to Quartus® II help.

**DDRIO Output Registers (altddio)**

For 32-bit transmitter cores, the altddio_out dual data rate (DDR) component is instantiated instead of the high-speed LVDS block. The instantiation within the core wrapper file is similar to the following code snippet:

```
aot1139_tx10_altddio_out altddio_out_dat (<signal
list> );
```

The aot1139_tx10_altddio_out module is defined within the same file as the core wrapper, and simply instantiates Altera's altddio_out component with the appropriate parameters. For details on the altddio_out component, refer to Quartus II help.

*Buffer Section*

The buffer section of the wrapper file includes a single buffer or multiple buffers, depending on the buffer mode selected when the core is configured. When configuring the MegaCore function, select from the following buffer options:

■ Individual FIFO buffer(s) mode—includes a separate buffer for each port (only option for single-PHY cores)

■ Shared FIFO buffer and embedded addressing mode—all ports to share a single buffer

■ Virtual FIFO segments per port and buffer management mode— divides a single memory into buffers for up to 16 ports

**Single-PHY Individual FIFO Buffer Mode**
The right side of Figure 2 on page 3 accurately reflects the structure of the buffer section of a single-PHY transmit MegaCore function. The Atlantic 'A' interface is the Atlantic-compliant interface connecting to your application-specific logic. You use this interface to push the data to be transmitted into the FIFO buffer. The Atlantic 'B' interface is the Atlantic-compliant interface between the buffer and the protocol core. The buffer is a dual-clock domain buffer, with the 'A' interface on the `a0_atxclk` clock domain and the 'B' interface on the `trefclk` clock domain, as identified by the dotted lines. This allows your application logic to run on a clock domain other than the POS-PHY Level 4 core's domain. The buffer instantiation in the single-PHY buffer mode is similar to the following code snippet:

```
aot1139_tx1_atltop aot1139_tx1_atlfifo_0( <signal
list> );
```

**Multi-PHY Individual FIFO Buffers Mode**
When the multi-PHY individual FIFO buffers mode is selected, a separate physical buffer is instantiated for each port. See Figure 3 on page 8. Each individual buffer is exactly the same as the buffer used in the single-PHY mode, and each has its own write-side clock domain. The underflow signal for each buffer is on the `trefclk` domain, while the overflow signal and error detection signals are on each buffer's clock domain.

*Figure 3. Individual FIFO Buffer(s) Structure*
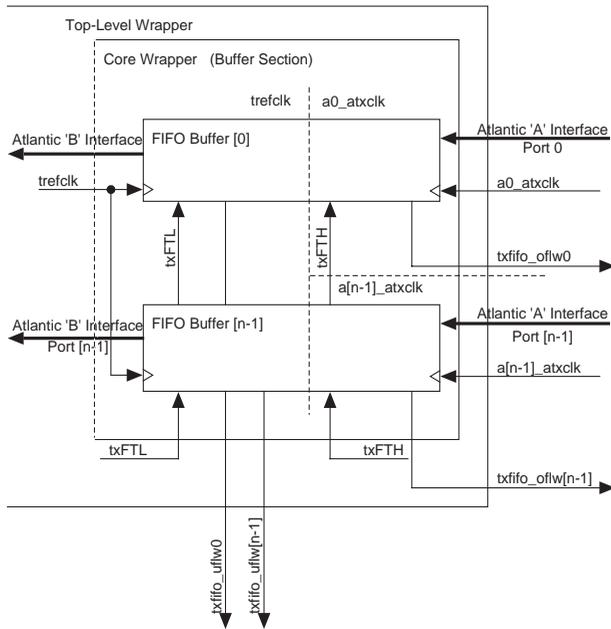


**Shared FIFO Buffer & Embedded Addressing Mode**

The shared FIFO buffer and embedded addressing mode, for multi-PHY cores, is nearly the same as the single-PHY mode. The only difference in the buffering scheme is that the port number is pushed into the buffer in parallel with the data, as shown in Figure 4 on page 9. The write side of the buffer uses a clock supplied by your application logic, whereas the core reads from the buffer using the transmit reference clock. The empty signal is on the buffer's read-side clock domain, while the overflow and error detection logic is on the write-side of the buffer.

*Figure 4. Shared FIFO Buffer with Embedded Addressing Structure*
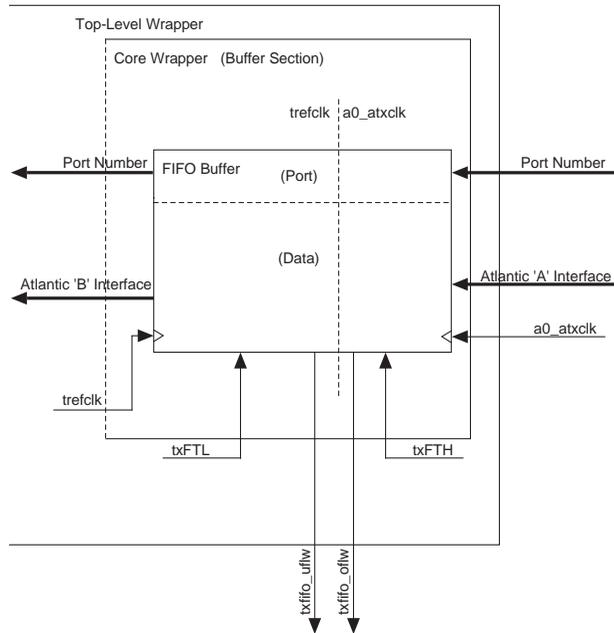


**Virtual FIFO Segments Per Port & Buffer Management Mode**
When the virtual FIFO segments and buffer management mode is
selected, a single block of memory is used as a storage area for all ports.
Your application logic must supply the port address when writing to the
buffer, as in the shared FIFO buffer with embedded addressing mode. The
core uses the port address to determine which segment of the shared
memory to write to. The protocol section of the core also provides the port
address to the buffer manager block when it reads from the buffer. The
buffer manager block determines the correct memory segment, and reads
data from the appropriate memory location. Figure 5 on page 10 shows a
block diagram of the virtual FIFO segments and buffer management
structure.

Refer to the *POS-PHY Level 4 MegaCore Function User Guide* for details
regarding the buffer manager status signals.

*Figure 5. Virtual FIFO Segments & Buffer Management Structure*
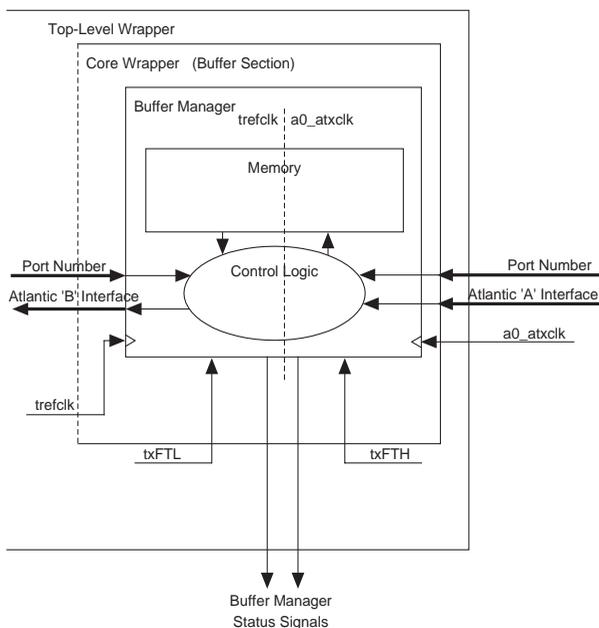


## Transmitter Core Wrapper Features

As stated previously, you can edit the top-level wrapper to modify the SPI-4.2 parameters without using the wizard, and you can connect the parameter inputs in the wrapper to registers so they can be run-time controlled by software. Similarly, you can modify the core wrapper to take advantage of additional features. The following subsections describe the features that are available by editing the transmitter's core wrapper file. All of the signals required to use these features are found in the core wrapper file under the comment *Optional Features Section*, with the exception of the INFO signals which are found just above that section.

Some additional features described within the wrapper file itself are only placeholders for features to be implemented in the future. These features include:

- DIP-4 error insertion
- Status channel active edge
- Atlantic FIFO error checking
- DIP-2 in-sync/out-of-sync thresholds

☞   These features are not supported and should not be used.

### INFO Signals

A number of signals—located just below the port declarations—are used to give information about the core. The INFO signals are statically-defined signals used to determine the MegaCore configuration that is in use. These signals can be displayed in a waveform simulation for reference, or can be tied to registers for software accessibility. Unless these signals are connected to a register or some other logic, the Quartus II software removes them during compilation. Table 3 shows an example set of INFO signals.

**Table 3. INFO Signal Examples**

| INFO signal name | Meaning |
|---|---|
| info_AOT = 1139 | This configuration is based upon build number 1139. |
| info_DFLOW_eq_TX | This core is a transmitter. |
| info_NPORTS_eq_4 | The core supports four ports. |
| info_DPATHW_eq_32 | The internal data path width is 32 bits. |
| info_BUFCFG_eq_SEPFIFO | A separate FIFO buffer is used for each port. |
| info_ATLDW_eq_32 | The Atlantic data width is 32 bits. |
| info_FDEPTH_eq_256 | The FIFO buffer depth is 256 elements (an element is 32 bits). |
| info_SEGDEPTH_eq_0 | The segment depth is zero. This only applies to the virtual FIFO segments and buffer management mode. |
| info_TXBOPT_eq_0 | The transmit bandwidth optimization feature is not enabled. |
| info_BRSTMODE_eq_0 | The burst mode feature is not enabled. |
| info_BRSTSIZE_eq_0 | The burst size is zero. This only applies when the burst mode feature is enabled. |

## FIFO Buffer Status Override

When you use a SPI-4.2 transmitter with the shared FIFO buffer with embedded addressing mode, you have the ability to control the transmission of data directly; rather than relying on the transmit scheduler included with transmitters using other buffering modes. When operating normally, the transmitter uses the decoded incoming status channel information and considers the worst case status to be the effective status for every port. For example, in a ten-port system, if one port indicates that it is satisfied, the scheduler considers all ten ports to be satisfied and does not transmit any data from the buffer because the data may contain bursts destined for the satisfied port. Similarly, if one port is hungry and the others are starving, the scheduler considers all ten ports to be hungry and directs the transmitter to transmit MaxBurst2 × 16 bytes.

You can override this behavior by asserting the `fifo_stat_override` signal, thus choosing to have the transmitter send data whenever it is available in the transmit buffer. If you choose this override feature, you must look at each port's `port_stat_n` status signal to determine whether or not data from that port should be pushed into the transmit buffer. Table 4 on page 12 describes the signals that you need to consider when using this feature.

☞     This feature is only available for the shared FIFO buffer and embedded addressing mode.

*Table 4. Fifo Status Override Signals*

| Signal Name | Description |
|---|---|
| `fifo_stat_override` | FIFO status override control signal<br>A 1-bit signal controlling how the transmitter schedules data to be sent.<br>● 1'b0: Normal operation. The transmitter considers the worst case status to be the effective status for every port<br>● 1'b1: The transmitter ignores the status message received from the receiver, and sends anything that is in the transmit buffer<br>When this signal is used, the application logic must interpret the status information and push data into the transmit buffer appropriately.<br><br>This signal should not be changed except when the core is in reset. Connecting this signal to a fixed value in the RTL results in a decrease in logic use because the synthesizer removes unnecessary logic.<br><br>This mode is only available for receiver cores with the shared FIFO buffer with embedded addressing mode. |
| `port_stat_0[1:0]`<br>`port_stat_1[1:0]`<br>.<br>.<br>.<br>`port_stat_n[1:0]` | Port status signals<br>A 2-bit indication of the status for each port, where the values are as follows:<br>● 2'b00: Starving<br>● 2'b01: Hungry<br>● 2'b10: Satisfied<br>● 2'b11: N/A<br>These signals are updated on the `trefclk` domain by the status receiver every time a new status message is received without a DIP-2 error.<br><br>These signals are always available, regardless of the state of `fifo_stat_override`. |

## Atlantic FIFO Buffer Performance

The Atlantic buffer performance feature allows you to change the internal structure of the FIFO buffers to achieve better performance. When set for higher performance, multiple memories are instantiated in parallel in order to increase timing margin. The `fifo_hspeed_setting` signal should be connected to a fixed value in the RTL code, so that the synthesis

tool can remove the unnecessary structures. Failure to connect this signal to a fixed value in the HDL will result in poor performance, and is not supported.

| Table 5. Atlantic Buffer Performance Signals | |
|---|---|
| **Signal Name** | **Description** |
| `fifo_hspeed_setting[1:0]` | FIFO performance control<br>A 2-bit signal to control the internal structure of the FIFO buffers, where the values are as follows:<br>● 2'b00: Single-memory instantiation. Default for 32- and 128-bit cores<br>● 2'b01: Dual-memory instantiation<br>● 2'b10: Quad-memory instantiation. Default for 64-bit cores<br>● 2'b11: Not Supported<br>This signal must be tied to a static value at synthesis time so that the Quartus II optimizer can remove the unneeded structures. Failure to do so will result in increased resource utilization and poor performance. |

### EOP-Based Data Available

In most applications where the POS-PHY Level 4 core is used, bursts are not scheduled to be transmitted until the data in the buffer is greater than the FIFO threshold low (FTL) setting. When this feature is enabled, bursts are also transmitted whenever there is a complete packet in the buffer. A complete packet is detected when an end of packet (EOP) marker is pushed into the buffer. The EOP-based data available feature is enabled by asserting the `atlantic_fifo_eopdav_enable` signal. See Table 6.

☞ This signal should not be changed for 32-bit cores, nor for cores that have the burst mode feature enabled.

| Table 6. EOP-Based DAV Signals | |
|---|---|
| **Signal Name** | **Description** |
| `atlantic_fifo_eopdav_enable` | EOP-based DAV enable<br>A 1-bit signal used to determine if data is sent when EOP markers are in the transmit buffer.<br>● 1'b0: Normal operation. Data is transmitted only when the FIFO buffer fill level is above FTL<br>● 1'b1: EOP-based operation. Data is transmitted when an EOP is pushed into the FIFO buffer, or when the FIFO buffer fill level is above FTL<br>This signal cannot be changed after the core has come out of reset. |

## Receiver Cores

Figure 6 shows the relationship between the top-level wrapper, the core wrapper, and the components that make up the MegaCore function, for a typical receiver. Instantiate the top-level wrapper in your SPI-4.2 design.

*Figure 6. Receiver Wrapper Structure*
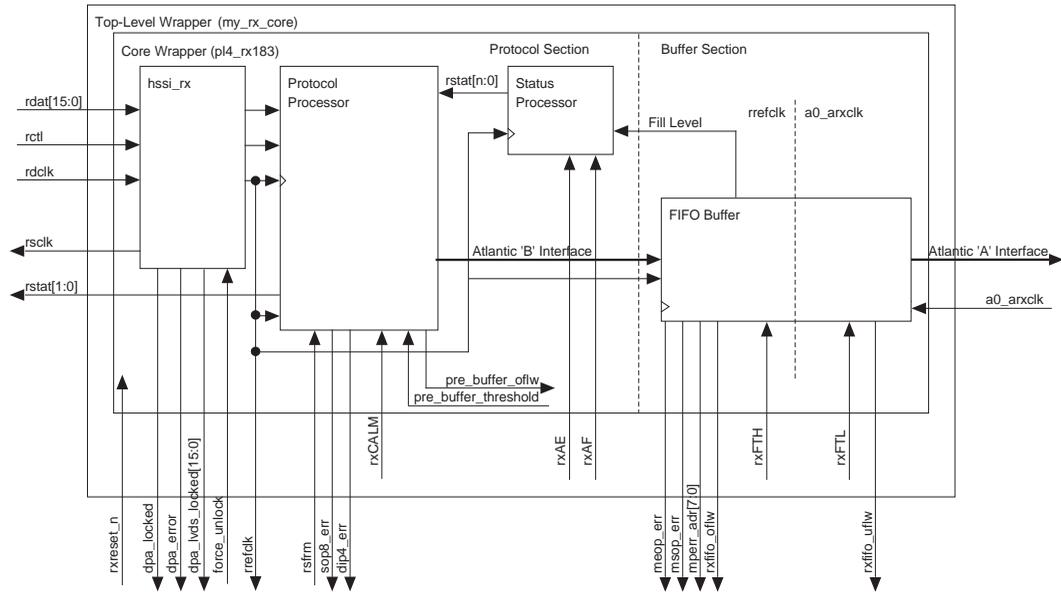


### Receiver Top-Level Wrapper Structure

The top-level wrapper is exceedingly simple. It does nothing other than to instantiate the core wrapper. All of the ports in and out of the core wrapper are passed through to be used by the application logic, with the exception of the parameters set by the wizard. The wizard writes these parameters (see Table 7 on page 15) into the top-level wrapper, based on the values you selected in the GUI to create the core. When writing these values, the wizard translates from units of bytes used in the GUI to units of FIFO elements used by the core, where FIFO elements are particular to the core type. (See Table 8 on page 16 for a summary.) Each parameter in the top-level wrapper has a corresponding signal named signal_wire*n*, where *n* is different for each parameter. By editing the wrapper file, you can change the values of these input signals to change the core parameters, or you can connect these signals to registers to allow software to change the core parameters during operation.

☞ Keep in mind that the units in the wrapper file are not the same as the units in the MegaWizard Plug-In.

**Table 7. Receiver Top-Level Wrapper Parameters**

| Name | Signal Name (1) | Description |
|------|-----------------|-------------|
| rxAE | signal_wire1[n:0] | The threshold (measured from the bottom of the FIFO buffer) below which a starving message is sent in the status message to the transmitter for each port. Changes to this parameter take affect immediately. *n* depends on the buffer depth. <br> **Range**: 1 ≤ rxAE ≤ FIFO buffer size <br> **Units**: FIFO elements <br> **Clock**: rrefclk |
| rxAF | signal_wire2[n:0] | The threshold (measured from the bottom of the FIFO buffer) above which a satisfied message is sent in the status message to the transmitter for each port. Changes to this parameter take affect immediately. *n* depends on the buffer depth. <br> **Range**: 1 ≤ rxAF ≤ FIFO buffer size <br> **Units**: FIFO elements <br> **Clock**: rrefclk |
| rxFTL (2) | signal_wire0[n:0] | The threshold (measured from the bottom of the FIFO buffer) used to indicate that there is enough data in the buffer to warrant that the transmitter start sending data to the application logic. Changes to this parameter take effect immediately. This should be set to a value of at least 16 bytes to maintain 16-byte transfers as required by the SPI-4.2 specification. *n* depends on the buffer depth. <br> **Range**: 0 ≤ rxFTL < FIFO size <br> **Units**: FIFO elements <br> **Clock**: Atlantic™ clock |
| rxFTH | signal_wire3[n:0] | Not used. |
| rxCALM | signal_wire4[7:0] | The number of times to repeat the status message before the DIP-2 code is sent. Changes to this parameter take effect immediately. This setting must match the transmitter's CALM setting. <br> **Range**: 1 ≤ rxCALM ≤ 0xFF <br> **Units**: Number of repetitions <br> **Clock**: rrefclk |

*Notes to Table 7:*
(1) The signal names are generated by the MegaWizard Plug-In.
(2) In the case of individual FIFO buffers, each buffer has its own Atlantic clock domain. If all of the buffers are driven from the same Atlantic clock, there is no problem. If the buffers are driven by different clocks, the rxFTL is sampled on different clock domains, thus the data available (dav) signal from the buffer may be incorrect for a number of clock cycles after rxFTL is changed. This has no effect on system performance.

*Table 8. Receiver FIFO Element Sizes*

| Receiver Data Path Width (bits) | Atlantic Data Width (bits) | Receiver FIFO Element Size (bytes) |
|---|---|---|
| 32 | 32 | 4 |
| 64 | 64 | 8 |
| | 128 | 16 |
| 128 | 128 | 32 |
| | 256 | |

## Receiver Core Wrapper Structure

The core wrapper is more complex than the top-level wrapper; it ties together the various blocks that make up the POS-PHY Level 4 MegaCore function. In addition, it includes extra features that can be accessed by editing the source file. This section describes the structure of the core, and how the components are wired together.

Figure 6 on page 14 shows how the core wrapper is divided into two sections: a protocol section and a buffer section. The protocol section includes the protocol processor, the status processor, and the high-speed serial interface (HSSI). The buffer section includes the FIFO buffer(s) and, depending on the configuration, demultiplexers to feed data from the protocol processor into the buffers. The details of the buffer section change drastically depending on the configuration.

### Protocol Section

The protocol section is responsible for handling everything related to the SPI-4.2 protocol.

### Protocol Core (ipcore)

The protocol core receives the SPI-4.2 data, translates and handles the control words, converts the packet to Atlantic format, and formats and sends the status message. The instantiation within the core wrapper is similar to the following code snippet:

```
aot1139_rx230_posphy4 ipcore(  <signal list> );
```

The *aot1139* name identifies the build of this MegaCore function; *rx230* is the configuration number. The configuration number may be different for every core with different wizard parameters.

**High-Speed Serial Interface (HSSI)**
The HSSI block receives the SPI-4.2 clock, the 16 LVDS data signals, and the single LVDS control signal. For 64- and 128-bit cores, the dedicated high-speed SERDES in the input buffers are used. For 32-bit cores, DDR input buffers are instantiated. For cores that include DPA circuitry, the DPA alignment and channel alignment circuitry is contained within this block. The instantiation within the core wrapper is similar to the following code snippet:

```
aot1139_rx288_hssi_rx hssi_rx( <signal list> );
```

**Status Processor**
The status processor block is a very simple block that determines the status of each of the ports in the system. It compares the fill level of each port's buffer to the almost empty (AE) and almost full (AF) thresholds defined for the system, and passes that information on to the protocol processor block for transmission.The instantiation within the core wrapper is similar to the following code snippet:

```
aot1139_rxstat_64_256_rxstatproc rxstatproc
( <signal list> );
```

## Buffering Section

The buffering section of the wrapper file includes a single buffer or multiple buffers, depending on the buffer mode selected when the core is configured. When configuring the MegaCore function, select from the following buffer options:

■ Individual FIFO buffer(s) mode—includes a separate buffer for each port

■ Shared FIFO buffer and embedded addressing mode—allows all ports to share a single buffer

■ Virtual FIFO segments per port and buffer management mode— divides a single memory into buffers for up to 16 ports

**Single-PHY Individual FIFO Buffer Mode**
The right side of Figure 6 on page 14 accurately reflects the structure of the buffer section of a single-PHY receive MegaCore function. The Atlantic 'A' interface is the Atlantic-compliant interface connecting to your application-specific logic. You use this interface to pull received data from the buffer. The Atlantic 'B' interface is the Atlantic-compliant interface between the buffer and the protocol core. The buffer is a dual-clock domain buffer, with the 'A' interface on the a0_arxclk clock

domain and the 'B' interface on the rrefclk clock domain, as identified by the dotted lines. This allows your application logic to run on a clock domain other than the POS-PHY Level 4 core's domain.

**Multi-PHY Individual FIFO Buffers Mode**
When the multi-PHY individual FIFO buffers mode is selected, a separate physical buffer is instantiated for each port. See Figure 7. Each individual buffer is exactly the same as the buffer used in the single-PHY mode, and each has its own read-side clock domain. The empty signal for each buffer is on the port's Atlantic clock domain, while the overflow signal and error detection signals are on the rrefclk clock domain.

*Figure 7. Individual FIFO Buffer(s) Structure*
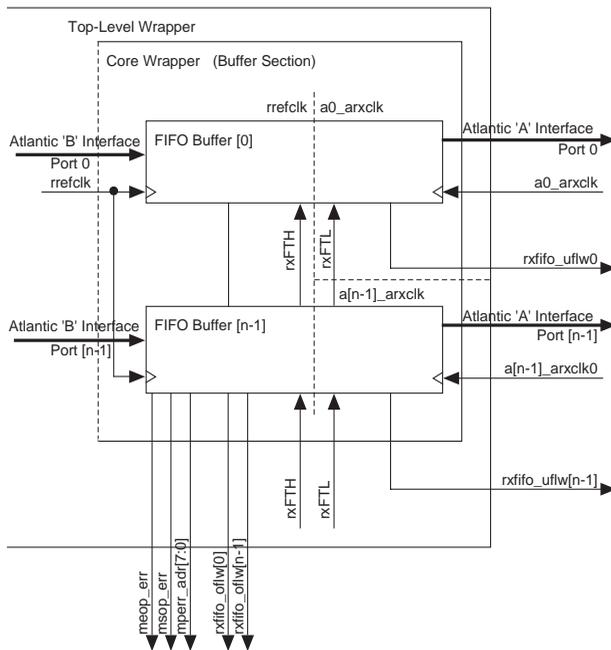


**Shared FIFO Buffer & Embedded Addressing Mode**
The shared FIFO buffer and embedded addressing mode for multi-PHY receiver cores is essentially the same as the one used for transmitter cores. The port number is simply pushed into the buffer in parallel with the data. See Figure 8 on page 19. The read side of the buffer uses a clock supplied by your application logic, whereas the core writing to the buffer uses the recovered receive clock. The empty signal is on the buffer's read-side clock domain, while the overflow and error detection logic is on the

write-side of the buffer. The Atlantic error detection logic resides on the write interface of the buffer, and includes the missing EOP and missing SOP output signals: rxmeop_err and rxmsop_err, respectively.

*Figure 8. Shared FIFO Buffer with **Embedded Addressing Structure***
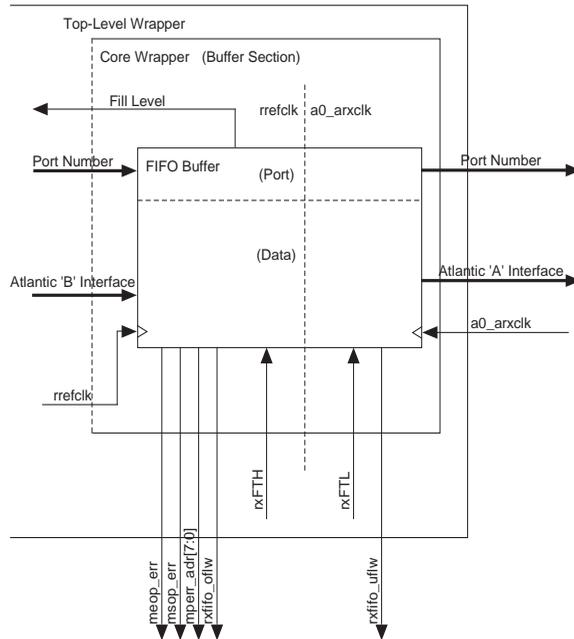


**Virtual FIFO Segments Per Port & Buffer Management Mode**
When the virtual FIFO segments and buffer management mode is selected, a single block of memory is used as a storage area for all ports. Your application logic must supply the port address when requesting data from the buffer. The core uses the port address to determine which segment of the shared memory to read from. The protocol section of the core also provides the port address to the buffer manager block when it writes to the buffer. Figure 9 on page 20 shows a block diagram of the virtual FIFO segments and buffer management structure.

Refer to the *POS-PHY Level 4 MegaCore Function User Guide* for details regarding the buffer manager status signals.

*Figure 9. Virtual FIFO Segments & Buffer Management Structure*
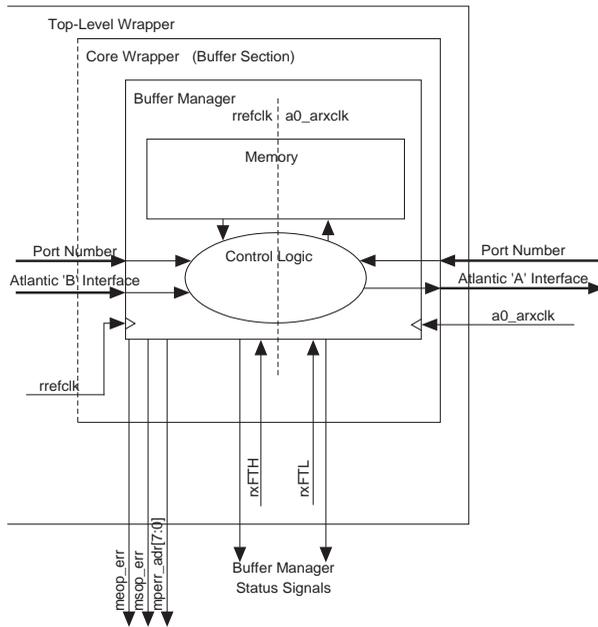


## Receiver Core Wrapper Features

As in the transmitter core, you can modify the core wrapper to take advantage of additional, available features. The following subsections describe the features that are available by editing the receiver's core wrapper file. All of the signals required to use these features are found in the core wrapper file under the comment *Optional Features Section*. The only exceptions are the INFO signals which are found just above that section.

Some additional features described within the wrapper file itself are only placeholders for features to be implemented in the future.

☞    These features, including: DIP-2 error insertion and status channel active edge, are not supported and should not be used.

*INFO Signals*

A number of signals—located just below the port declarations—are used to give information about the core. The INFO signals are statically-defined signals used to determine the MegaCore configuration that is in use. These signals can be displayed in a waveform simulation for reference, or can be tied to registers for software accessibility. Unless these signals are connected to a register or some other logic, the Quartus II software removes them during compilation. Table 9 shows an example set of INFO signals.

**Table 9. INFO Signal Examples**

| INFO signal name | Meaning |
|---|---|
| `info_AOT = 1139` | This configuration is based upon build number 1139. |
| `info_DFLOW_eq_RX` | This core is a receiver. |
| `info_NPORTS_eq_4` | The core supports four ports. |
| `info_DPATHW_eq_32` | The internal data path width is 32 bits. |
| `info_BUFCFG_eq_SEPFIFO` | A separate FIFO buffer is used for each port. |
| `info_ATLDW_eq_32` | The Atlantic data width is 32 bits. |
| `info_FDEPTH_eq_256` | The FIFO buffer depth is 256 elements (an element is 32 bits). |
| `info_SEGDEPTH_eq_0` | The segment depth is zero. This only applies to the virtual FIFO segments and buffer management mode. |
| `info_TXBOPT_eq_0` | The transmit bandwidth optimization feature is not enabled.This only applies to transmitters. |
| `info_BRSTMODE_eq_0` | The burst mode feature is not enabled. This only applies to transmitters. |
| `info_BRSTSIZE_eq_0` | The burst size is zero. This only applies to transmitters. |
| `info_MCONT_eq_1` | The multiple continue support feature is enabled. |

### FIFO Buffer Status Override

When you use a SPI-4.2 receiver with the shared FIFO buffer with embedded addressing mode, you can control the status channel directly; instead of having the protocol core compute it based on the fill levels of the buffer. When operating normally, the receiver sends the same status information for every channel, depending on whether the shared FIFO buffer with embedded addressing is above the AF threshold, between AE and AF, or below the AE threshold.

You can override this behavior by asserting the use_external_status signal, thus choosing to control the information sent on the status channel directly. When the use_external_status signal is asserted, you must

drive the `ext_stat_n` signal for each port to the appropriate 2-bit level, as defined in the SPI-4.2 specification. The protocol core then sends the status information out of the core directly, ignoring the fill level of the shared FIFO buffer with embedded addressing. Table 10 describes the signals you will need to manipulate when using this feature. These signals can all be found in the core wrapper file. You must edit this file to make the signals available at the top level.

☞ This feature is only available for the shared FIFO buffer and embedded addressing mode.

*Table 10. FIFO Status Override Signals*

| Signal Name | Description |
|---|---|
| `use_external_status` | FIFO status override control signal<br>A 1-bit signal controlling how the receiver sends the status channel.<br>● 1'b0: Normal operation.The receiver uses the status of the shared FIFO buffer with embedded addressing as the status for every port<br>● 1'b1: The receiver ignores the fill level of the shared FIFO buffer with embedded addressing, and instead sends the status supplied by the application logic<br>This signal should not be changed except when the core is in reset. Connecting this signal to a fixed value in the RTL results in a decrease in logic use because the synthesizer removes unnecessary logic.<br>This mode is only available for receiver cores with the shared FIFO buffer with embedded addressing mode. |
| `ext_stat_0[1:0]`<br>`ext_stat_1[1:0]`<br>.<br>.<br>.<br>`ext_stat_n[1:0]` | Port status signals<br>A 2-bit indication of the status to send for each port, where the values are as follows:<br>● 2'b00: Starving<br>● 2'b01: Hungry<br>● 2'b10: Satisfied<br>● 2'b11: N/A<br>These signals need to be updated by the application logic on the `rrefclk` clock domain.<br>These signals are only used when the `use_external_status` signal is asserted high. |

*Atlantic FIFO Buffer Performance*

The Atlantic buffer performance feature allows you to change the internal structure of the FIFO buffers to achieve better performance. When set for higher performance, multiple memories are instantiated in parallel in order to increase timing margin. The `fifo_hspeed_setting` signal

should be connected to a fixed value in the RTL code, so that the synthesis tool can remove the unnecessary structures. Failure to connect this signal to a fixed value in the HDL will result in poor performance.

*Table 11. Atlantic Buffer Performance Signals*

| Signal Name | Description |
|---|---|
| `fifo_hspeed_setting[1:0]` | FIFO performance control<br>A 2-bit signal to control the internal structure of the FIFO buffers, where the values are as follows:<br>● 2'b00: Single memory instantiation. Default for 32-and 128-bit cores<br>● 2'b01: Dual-memory instantiation<br>● 2'b10: Quad-memory instantiation. Default for 64-bit cores<br>● 2'b11: Not Supported<br>This signal must be tied to a static value at synthesis time so that the Quartus II optimizer can remove the unneeded structures. Failure to do so results in increased resource utilization and poor performance. |

*EOP-Based Data Available*

In most applications where the POS-PHY Level 4 core is used, data is not forwarded to the application logic until the data in the buffer is greater than the FTL setting. This is to allow the application logic to handle data more efficiently by limiting the frequency of port switches. When this feature is enabled, the data available (arxdav) signal is also asserted to the application logic whenever there is a complete packet in the buffer. A complete packet is detected when an end of packet (EOP) marker is pushed into the buffer by the receiver. The EOP-based data available feature is enabled by asserting the atlantic_fifo_eopdav_enable signal. See Table 12.

*Table 12. EOP Based DAV Signals*

| Signal Name | Description |
|---|---|
| `atlantic_fifo_eopdav_enable` | EOP-based DAV enable<br>A 1-bit signal used to determine if data is sent when EOP markers are in the transmit buffer.<br>● 1'b0: Normal operation. Data is transmitted only when the FIFO buffer fill level is above FTL<br>● 1'b1: EOP-based operation. Data is transmitted when an EOP is pushed into the FIFO buffer, or when the FIFO buffer fill level is above FTL<br>This signal cannot be changed after the core has come out of reset. |

*Pre-Buffer Control & Status*

All 128-bit receiver cores with the `multiple continue` parameter enabled, and all 64-bit receiver cores, include a 256-element deep FIFO pre-buffer. The pre-buffer allows the core to buffer data in the event that more than one internal `rrefclk` clock cycle is required to properly process the incoming SPI-4.2 data. Although this pre-buffer normally operates in the empty or near empty state, a threshold is provided to prevent overflows. When the fill level of the FIFO pre-buffer exceeds the threshold setting, the core provides backpressure by sending a satisfied status on all ports. This prevents the transmitter from sending excessive data while the FIFO buffer is being cleared out. Once the pre-buffer fill level empties, the status message sent by the receiver returns to indicating the status of the Atlantic FIFO buffer(s). In the event that the pre-buffer overflows, the core notifies the transmitter by sending a framing signal on the status bus. When the pre-buffer fill level empties, the receiver's status returns to indicating the status of the Atlantic FIFO buffer(s).

In the core wrapper file, you have the ability to change the threshold setting used to determine when to send backpressure to the transmitter. You also have access to the pre-buffer overflow signal. See Table 13.

| Table 13. Pre-Buffer Control & Status | |
|---|---|
| **Signal Name** | **Description** |
| `pre_buffer_threshold[7:0]` | Pre-buffer threshold<br>An 8-bit signal defining the pre-buffer's almost full setting. When the fill level of the pre-buffer is above this setting, the receiver backpressures the transmitter by sending a satisfied status message on all ports. It returns to normal operation after the pre-buffer drains to empty.<br>This signal cannot be changed after the core has come out of reset. |
| `pre_buffer_oflw` | Pre-buffer overflow<br>This status signal is asserted high in the event that the pre-buffer has overflowed. This indicates a loss of data and a possible system configuration problem. |

*Atlantic FIFO Error Checking*

By default, the receiver core includes logic to check for a missing start of packet (`MSOP`) and missing end of packet (`MEOP`) on each packet received. These error conditions are marked with the `msop_err` and `meop_err` output pins, and cause the Atlantic error signal to be asserted with the next `EOP` indication for that port, allowing downstream logic to drop the errored packet. The core wrapper includes the ability to disable this error-checking functionality, saving resources. This resource savings becomes considerable for systems with a large number of ports.

It is important to recognize that if this error checking functionality is disabled, a packet with a missing SOP or EOP will be silently merged with the preceding or following packet, resulting in errored packets that are not marked as errored.Therefore, you should not disable the error checking unless you are confident that the system design precludes the occurrence of missing SOP and EOP markers, or that there is a higher-level packet integrity mechanism (such as a CRC checker) that verifies that packets are not corrupted.

To disable the error checking in the receiver, connect the errchk_checkpkt signal in the core wrapper to a constant zero value, and let the synthesizer remove the unneeded logic.

| Table 14. Atlantic FIFO Error Checking | |
|---|---|
| **Signal Name** | **Description** |
| errchk_checkpkt | Atlantic FIFO error checking<br>A 1-bit signal to enable or disable the MSOP and MEOP checking in the receiver's FIFO buffer.<br>● 1'b1: Normal operation. Error checking is enabled, and errored packets are marked with the Atlantic error signal<br>● 1'b0: Error checking is disabled, and MSOP and MEOP errors go undetected<br>This signal must be tied to a static value at synthesis time so that the Quartus II optimizer can remove the unneeded structures. Failure to do so results in increased resource utilization and poor performance. |

## References

The following documents should be consulted when using the POS-PHY Level 4 MegaCore function.

■ Altera, *POS-PHY Level 4 MegaCore Function User Guide*
■ Optical Internetworking Forum (OIF), *System Packet Interface Level 4 (SPI-4) Phase 2: OC-192 System Interface for Physical and Link Layer Devices*