Opal Kelly

FrontPanel™

A new way to control and observe FPGA designs
through virtual instruments on your PC.

Opal Kelly's FrontPanel software is designed to provide controllability and observability for FPGA designs. It's unique design allows users to describe their own control panels using industry-standard XML descriptions of components such as LEDs, hex displays, push buttons, toggle buttons, triggers, and so on. The components then connect to endpoints within the user's FPGA design. Once connected, the interface details are transparent. FrontPanel handles all interaction between the virtual controls and the FPGA internals. In the end, FrontPanel eliminates the time and effort of interfacing to a design and greatly assists in the external controllability and observability of that design.

Revision History:

Date	Description
20101201	Initial release with both USB and PCI Express.
20101211	Fixed USB BTPipeOut timing diagram.
20111221	Added note about 32-bit and 64-bit API usage.
20120103	Added QNX application note.
20120313	Updated PCIe performance metrics.
20120608	Updates with USB 3.0.
20121002	Fix typo in QNX section.
20121003	FrontPanel Interface Update.
20121112	Additional information about wrapper APIs.
20130117	Fixed direction in the USB 3.0 okRegisterBridge pinout table.
20130409	Removed references to GetDeviceListID().
20130810	Added command line arguments.
20130918	Added documentation for okFilePipe blocksize parameter.
20131114	Additional information on Registers API. Added Device Sensors and Device Settings APIs.
20140505	Change references to okFrontPanelDLL.cpp to the new import library style.
20140808	Fix several typos.
20141220	Remove reference to okFrontPanel.cpp.
20150120	Added new Simulation Sample.
20150126	Updated 32-bit / 64-bit architecture notes.
20150309	Updated FrontPanel Application notes and screenshots.

Contents

An Introduction to FrontPanel

FrontPanel is a software platform designed to make FPGA integration easier, more productive, more powerful, and more configurable. Most importantly, FrontPanel provides the basic functionality required to configure and interface to the hardware including the FPGA and peripherals on-board. After FPGA configuration, the host interface (USB or PCI Express) switches from a high-speed download port to active communication with FrontPanel allowing you to interface and control your FPGA design from within a single application. By virtualizing many common controls found on typical evaluation (or prototyping) boards, FrontPanel enables far greater flexibility and capability than pure hardware-based approaches.

The FrontPanel SDK (Software Development Kit) is a flexible API (Application Programmer's Interface) providing all the benefits of FrontPanel to your own custom application. These benefits include:

- Device discovery and enumeration
- FPGA configuration
- FPGA communication using wires, triggers, pipes
- Abstraction to a common development platform for both USB and PCI Express devices

Terminology

Collectively, "FrontPanel" describes several components that make up the FrontPanel environment:

- "FrontPanel SDK" - Refers to the software development kit - the HDL, API, and documentation collectively.

- "FrontPanel HDL" - HDL modules you design into your FPGA hardware that makes your design "FrontPanel Enabled" and allows it to communicate with the PC.
- "FrontPanel Firmware" - Firmware running on the module's microcontroller that provides the conduit for FPGA/PC communication.
- "FrontPanel API" - A complete programmer's interface allowing you to design custom PC applications that communicate with your FrontPanel Enabled hardware.
- "FrontPanel Application" - A flexible software application providing virtual instrumentation to your hardware such as LEDs, hex displays, numeric entry, pushbuttons, and so on.

Basic Functionality

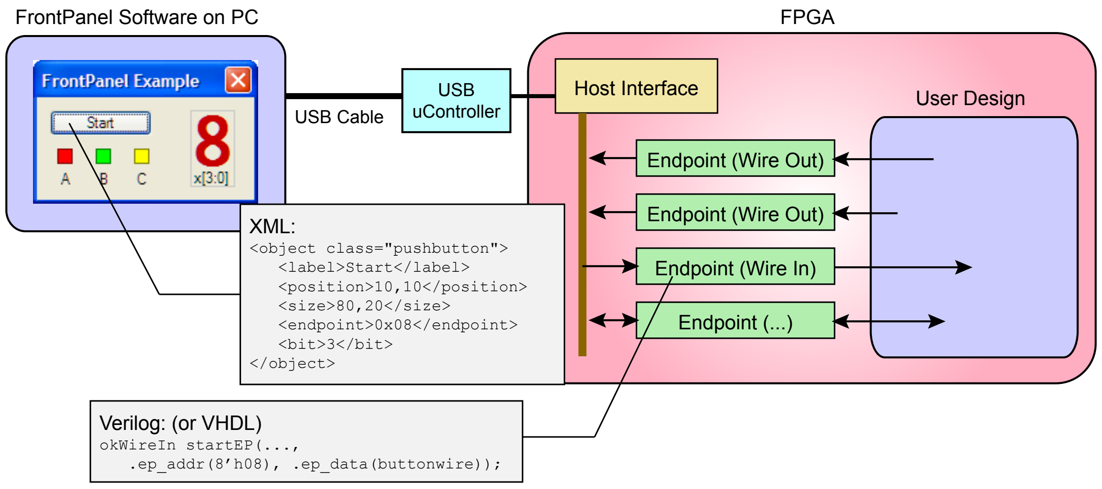
FrontPanel is, most importantly, support software for Opal Kelly's FPGA integration modules. In that role, FrontPanel allows you to quickly and easily download FPGA configuration files via USB or PCI Express to a target device. Once the configuration file is downloaded, the device now takes on that design's personality and is ready for use. If desired, FrontPanel's role is now complete.

Peripheral Configuration

Opal Kelly XEM devices contain additional peripherals to integrate FPGAs into your projects. PLLs, audio CODECs, Flash memory, and other peripherals can benefit from the simple, single-source configuration that FrontPanel offers. PLL outputs are independently configurable through easy-to-use setup dialogs. Flash memory can be programmed, cleared, and reprogrammed in a variety of ways and audio CODECs can be setup for different configurations.

Flexibility Outside the Design

FPGAs and other programmable logic devices have allowed engineers the unique opportunity to construct complete hardware designs within the confines of a logic device. Unfortunately, this means that many of the tools engineers typically employ to debug such designs such as oscilloscopes, LEDs, switches, and buttons are limited to viewing signals brought out to the external pins of the device. FrontPanel takes these ideas closer to the realm of flexible logic devices and makes them likewise flexible. In the end, however, FrontPanel provides controllability and observability to your designs, reducing development time and putting a new face on your prototypes.



```
<object class="pushbutton">
   <label>Start</label>
   <position>10,10</position>
   <size>80,20</size>
   <endpoint>0x08</endpoint>
   <bit>3</bit>
</object>
```

```
okWireIn startEP(...,
   .ep_addr(8'h08), .ep_data(buttonwire));
```

Controllability

Any prototype or experiment invariably requires some level of control. Typically, devices such as pushbuttons, DIP switches, rotary devices, or keypads are used. With most prototype systems, however, what is provided is nearly never enough; you can always use one more button or switch to select a different mode. Typically, the problem is solved by rebuilding the design with a different mode or multiplexing the use of the available inputs.

FrontPanel offers another solution. With a simple change in a couple files, new virtual buttons and switches can be added quickly and connected to the proper points in your design.

Observability

Prototypes also require some level of observability, usually offered in the form of LEDs, hexadecimal displays, and LCDs or sampled externally by oscilloscopes and logic analyzers. Again, however, there is the problem of limited resources in the typical prototype system. Only so many LEDs and displays are present on an I/O board, so the problem is remedied by adding more I/O boards or multiplexing the use of the current lot.

FrontPanel's flexibility means you can display all sorts of information, in real-time, about the state of any number of signals in your design. It's like having an I/O board that allows you to add and remove components at will without taking up valuable pins on the FPGA.

XML and FrontPanel Components

XML is the eXtensible Markup Language used in the latest generation of software applications and other forms of markup (such as XHTML). It is simply a way to describe data that can be manipulated by any XML-supporting editor and in a platform-agnostic way. At its core, XML is just a text file containing tags which correspond to nodes of a tree. Each node can have properties and values.

FrontPanel interfaces are described using XML tags so they can be read and written with any standard text editor. This means that adding components to your virtual "I/O board" is as easy as adding a few lines to a text file. It also means that as FrontPanel grows in its capabilities, the interface descriptions will be forward (and backward) compatible. As additional functionality is added to FrontPanel, you will be able to take advantage of it by simply adding to your current projects.

HDL Endpoints

On the FPGA side of the interface, "Endpoints" are used to connect FrontPanel components to signals in your design. These endpoints work just like any external pin. You simply connect the signals you want to control or observe to the endpoint ports. Then, connect the endpoint modules to a shared bus and place a Host Interface module on that same shared bus. The Host Interface along with FrontPanel software and drivers take care of the rest. Signals within the FPGA are immediately visible within FrontPanel and FrontPanel can now control any input endpoints you've connected.

Additional endpoints can be added at any time simply by instantiating additional endpoint modules. The modules are designed to consume very little FPGA resources so the effect on your design is minimal.

Application Programmer's Interface (API)

Beyond the relatively basic instruments available within the FrontPanel Application, programmers can communicate with their HDL endpoints from their own software. In addition to the wires and triggers that populate a FrontPanel XML user interface, your custom applications can easily send and retrieve bulk data at HighSpeed USB or PCI Express throughput. This functionality broadens the application base of Opal Kelly integration modules to areas such as image capture, data acquisition, FPGA co-processing, signal generation and many others.

The majority of Opal Kelly integration applications involve the use of the API in some form. Customers use the API on Windows, Linux, and Apple (Mac) platforms. Some customers use our DLL directly within their C or C++ application. Some use our pre-built wrappers for C#, Python, Java, and Ruby. Others build wrappers for their own languages such as VisualBasic. Still others use our DLL from within third-party application software such as Matlab or LabVIEW.

Designing with FrontPanel

FrontPanel's main purpose is to move data between your PC and your FPGA in order to provide a convenient and effective way for you to work with the design. FrontPanel was designed to interface simply and easily with new and existing FPGA designs in a way which is powerful enough to apply to a large number of interface methods, yet simple enough to apply to a design in minutes. More importantly, FrontPanel attempts to make the specific implementation of the physical interface (USB or PCI Express, depending on your device) disappear so that those details don't get in the way of your work.

FrontPanel introduces the concept of "endpoints" to your FPGA design. An endpoint is a bundle of interconnect internal to your design that transports data to or from the PC in some fashion. In many cases, the endpoint can be created from an existing signal in your design which you want to observe in FrontPanel. In other cases, you will create an endpoint to perform a specific data transfer.

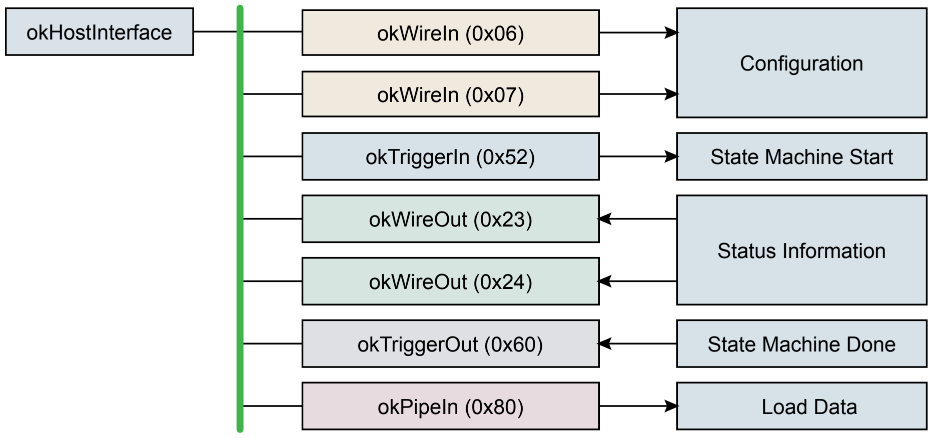
When using the FrontPanel Application, "Components" are the corresponding PC-side interface to an endpoint in the FPGA. Components may correspond to a single bit on an endpoint or to several endpoints. For example, an okTriggerButton activates a single bit on a Trigger In endpoint. In contrast, a field that allows you to enter or display a number spanning more than 256 would map to multiple endpoints.

When using the FrontPanel SDK in your own application, the API methods are the corresponding PC-side interface to an endpoint in the FPGA.

Endpoints

In FrontPanel, an endpoint is either a Wire, Trigger, or Pipe, and is either directed in or out of your design. By way of definition, the endpoint will always be labelled from the perspective of the device (FPGA) so an "In" endpoint moves data into the design while an "Out" endpoint moves data out of the design. All of the endpoints in a design are instantiated from Opal Kelly modules and share a common connection to the Host Interface which provides the connection to the PC through the USB or PCIe interface on the XEM board.

The figure below shows the block diagram of an example FPGA design. The okHostInterface is instantiated once and connects to the external FPGA pins as well as a bus shared by all endpoint HDL modules. This bus provides the communications channel for the endpoints to and from the Host Interface.



Each instance of an endpoint has an associated address (shown in parentheses) so it may be accessed independent of other endpoints. In this example, two Wire In endpoints setup the configuration for the design and two Wire Out endpoints relay status information back to the PC. The Trigger In endpoint is used to initiate a state machine and a Trigger Out endpoint is used to indicate the completion of the state machine. A Pipe In endpoint is used to load data into a memory within the design.

The three types of endpoints are summarized in the table below and described in more detail after.

Endpoint	**Sync/Async**	**Description**
Wire In	Asynchronous	Transfers a signal state into the design. (Examples: virtual pushbutton or switch)
Wire Out	Asynchronous	Transfers a signal state out of the design. (Examples: virtual LED or hex display)
Trigger In	Synchronous	Generates one-shot signal destined for a particular clock. (Example: pushbutton to start a state machine)
Trigger Out	Synchronous	Informs the PC that a particular event has occurred. (Example: Done signal from a state-machine pops up a window to the user or starts a data transfer)
Pipe In	Synchronous	Multi-byte synchronous transfer into the design. (Example: Memory download, streaming data)
Pipe Out	Synchronous	Multi-byte synchronous transfer out of the design. (Example: Memory upload, read results of a computation)

Wires

A Wire is an asynchronous connection between the PC and an HDL endpoint. A Wire In is an input to the target. A Wire Out is an output from the target.

Wires are designed to fill the position of devices such as LEDs, hexadecimal displays, pushbuttons, DIP switches, and so on. These devices are not synchronous to the design and they usually convey the current state of some internal signal (in the case of Wire Outs).

Wires are updated periodically using a polling mechanism. The rate of update is determined by how fast the PC can poll the FPGA. In FrontPanel, this value is user-configurable. Even at the highest update rate (25 millisecond period), very little bandwidth is consumed, so you should not notice any performance penalty.

Because some FrontPanel components may convey the state of several wires, and in order to avoid multiple transfers over the bus, all wires are captured and updated simultaneously. That is not to say they are synchronous, but that they are all updated at the same time. Therefore, all 64 Wire Ins (or Wire Outs) are transferred together.

Triggers

Triggers are synchronous connections between the PC and an HDL endpoint. A Trigger In is an input to the target. A Trigger Out is an output from the target. Triggers are used to initiate or signal a single event such as the start or end of a state machine.

As an input to the HDL, a Trigger In creates a signal that is asserted for a single clock cycle. The synchronization clock is determined by the user and the HDL module takes care of crossing the clock domains properly.

As an output from the HDL, a Trigger Out triggers the PC when a signal's rising edge is detected. The "rising edge" is actually determined by the signal's state from one clock cycle to the next and does not detect glitches. It should be noted that because FrontPanel polls the FPGA periodically, it can only detect independent trigger outs between polls. That is, once a Trigger Out is "set," it remains set until the next poll clears it.

Pipes

Pipes are synchronous connections between FrontPanel and an HDL endpoint. Unlike Triggers which convey a single event, however, Pipes are designed to transmit a series of bytes to (or from) the endpoint. They are most commonly used to download or upload memory contents but may also be used to stream data to or from the device.

From the HDL point-of-view, a Pipe is always a master. That is, the PC (and therefore the HDL module that implements the Pipe) controls the transaction for both Pipe Ins and Pipe Outs. In addition, the Pipe transactions must be performed at the endpoint's clock rate (48 MHz for USB devices, 50 or 100 MHz for PCIe devices). To reliably cross this clock boundary, a buffered (FIFO) arrangement is suggested. The Xilinx Core Generator can produce an appropriate FIFO for you.

Although access to the Pipe is always from a slave point of view, use of Triggers provides an effective negotiation method to synchronize the transfer of blocks of data.

Pipe transfer rates will vary depending on host hardware. Our tests indicate transfer rates up to 38 MB/s for USB 2.0 devices, 200 MB/s for PCIe devices, and over 300 MB/s for USB 3.0 devices. For more detail, see Performance Notes below.

FrontPanel-3 Note

Firmware supporting FrontPanel 1.4.1 and earlier was limited to approximately 32 MB/s to the FPGA and 19 MB/s from the FPGA.

Block-Throttled Pipes

Block-Throttled Pipes (or BTPipes) are very similar to "standard" Pipes with one important distinction: BTPipes provide a way for the FPGA to "throttle" transfer through the pipe at a block level. The block size is programmable from 1 to 512 words (2 to 1024 bytes). The FPGA throttles data through the BTPipe by asserting or deasserting a READY signal to the USB microcontroller. This allows the FPGA to halt data transfer until data is available or ready to be processed.

BTPipes provide the same transfer rates as standard pipes, but the throttling allows them to be used in a wider array of applications and can, generally, increase performance by reducing the overhead that would otherwise be required to negotiate the transfer at a higher level.

Block-Throttled Pipes are treated as standard Pipes on PCI Express devices.

FrontPanel-3 Note

BTPipes are only available using firmware supporting FrontPanel-3.

Full-Speed USB Note

On full-speed USB busses, the block size is limited to 1 to 32 words (2 to 64 bytes).

Registers

The USB 3.0 implementation of FrontPanel includes a "register bridge" that provides an addressable read/write register access point to customer HDL. The interface includes a read strobe and write strobe as well as 32-bit address and 32-bit data ports. This allows the host to access a 16GB addressable register range in the user HDL.

Components

Components represent the other half of the interface, each connecting to an appropriate endpoint or multiple endpoints within the design. Most components have a graphical representation within FrontPanel such as a pushbutton, virtual LED, or numerical display. Some components, however, are hidden from view. An example of a hidden component would be one that makes a sound in response to a Trigger Out.

Performance Notes

Opal Kelly's FrontPanel consists of HDL modules within the FPGA, firmware on the USB microcontroller (or PCIe bridge device), and an API on the PC that have been optimized for both performance and a clean abstraction. Our latest FrontPanel-3 release has improved performance significantly while offering several features that customers have requested.

Achieving the highest level of performance for your particular application requires an understanding of the components being used and how certain things affect performance. By following a few simple strategies and applying these notes, your application will be a top performer and still benefit from the ease of use and flexible abstraction that only FrontPanel provides.

Measured Performance

Measured performance figures in this section were taken on an Athlon 64 X2 4800+ machine running Windows XP SP2. USB performance can vary significantly depending on a number of factors including the motherboard make and model, specific driver versions installed, and machine load. The PipeTest application can be used as a simple benchmark.

Wires and Triggers

Wires and triggers provide the most basic form of communication between the FPGA and the PC. From a performance perspective, wires can be read or written several hundred times per second. All WireIns are read simultaneously, regardless of which ones you are interested in. Similarly, all WireOuts are written simultaneously.

Activating a TriggerIn is a very fast operation and can operate at over 1,000-times per second. Only one trigger is written per call. Updating TriggerOuts is similar to reading all WireOuts: all TriggerOuts are read simultaneously.

Since Wire and Trigger updates are always blocking API calls, these measurements provide some indication of the latency performance of the device.

Measured Performance (CPS = Calls Per Second)

API Call	USB 3.0 (CPS)	USB 2.0 (CPS)	PCIe (CPS)
UpdateWireIns	5,000+	1,000+	4,000+
UpdateWireOuts	4,000+	800+	3,000+
ActivateTriggerIn	8,000+	2,000+	66,000+
UpdateTriggerOuts	4,000+	800+	3,000+

Pipes (Bulk Transfers)

Pipes are the fastest way to transmit or receive bulk data. Due to overhead, performance is best with long transfers. Each time you perform a pipe transfer, several layers of setup are required including those at the firmware level, API level, and operating system level. Therefore, it is best to design around using long transfers, if possible. This generally means using large buffer sizes on the FPGA and relying on external memory when possible.

Low-latency, high-bandwidth transfers present a special challenge to any protocol and USB (and therefore FrontPanel) is no different. In this case, the two goals are at odds: trying to perform many operations and still achieve high bandwidth. The problem is that the overhead associated with setting up each transfer cuts into the time available to perform the data transfer.

It is important to note that Windows, Linux, and Mac OS X are not real-time operating systems. They are complex systems that may have many other processes taking higher priority at any given time. Therefore, it is often the case that simple operations (like moving a window) dramatically reduce transfer bandwidth. This should be a consideration when designing the buffering for any bandwidth-dependent application.

NOTE: Pipes in FrontPanel-3 are actually a subset of Block-Throttled Pipes where the `EP_READY` signal is always asserted, thus disabling any throttling. Also, block sizes are always 1024 bytes except for the last block which may be smaller to account for the total length of the transfer. Block sizes are 64 bytes when the device is enumerated at full-speed.

Measured Performance

All values in MB/s (megabytes per second). Writes measured with WriteToPipeIn. Reads measured with ReadFromPipeOut.

	USB 3.0		USB 2.0		PCI Express	
Transfer length	**Write**	**Read**	**Write**	**Read**	**Write**	**Read**
128 bytes	0.06	0.12	0.100	0.100	TBD	TBD
256 bytes	0.12	0.24	0.100	0.200	TBD	TBD
512 bytes	0.24	0.49	0.300	0.400	TBD	TBD
1.0 kB	0.49	0.98	0.700	0.800	16.1	15.8
4.0 kB	0.98	3.91	2.8	3.1	58.7	66.6
16.0 kB	7.81	15.6	8.9	10.4	100	125
64.0 kB	31.3	55.0	20.8	23.2	100	172
256 kB	125	150	31.8	32.7	100	185
1.0 MB	252	258	36.5	36.7	100	200
4.0 MB	313	313	37.9	37.9	100	200
8.0 MB	321	318	38.2	38.1	100	200

Block-Throttled Pipes (Bulk Transfers)

Block-Throttled Pipes are available only in FrontPanel-3 implementations on USB devices. They provide equivalent performance to the standard pipe except that the FPGA can throttle the data transfer at the block level. The block is programmable by the user with highest performance achieved at the largest (1,024-byte) block size.

BTPipes are an excellent way to achieve high performance with smaller buffer sizes because the FPGA can negotiate the transfer at a low level without incurring the significant overhead of setting up a new transfer for each small buffer block.

Measured Performance

All measurements taken with a 8-MB transfer length.

Block length (bytes)	WriteToBlockPipeIn	ReadFromBlockPipeOut
4	353 kB / s	266 kB / s
16	1.33 MB / s	1.03 MB / s
64	4.88 MB / s	3.98 MB / s
256	17.7 MB / s	14.0 MB / s
300	20.6 MB / s	13.8 MB / s
400	24.8 MB / s	16.9 MB / s
512	29.9 MB / s	24.5 MB / s
600	32.8 MB / s	21.9 MB / s
700	35.1 MB / s	22.4 MB / s
800	35.7 MB / s	23.0 MB / s
900	35.0 MB / s	22.7 MB / s
1024	38.2 MB / s	38.1 MB / s

Isochronous Transfers?

FrontPanel does not support USB isochronous transfers. It is true that isochronous transfers can negotiate for guaranteed bandwidth on the USB which can be very helpful when trying to build a system that must deliver certain performance to the end-user. However, this guarantee comes at a significant price: isochronous transfers do not provide the same level of error-detection and error-correction that the more reliable USB bulk transfers provide. Furthermore, the guarantee is only for bus bandwidth and says nothing about the operating system's capabilities.

If an error occurs during the transmission of a bulk transfer, the host will request that the missing packet be repeated. The host will also properly reconstitute the transmission so that everything is properly sequenced.

With isochronous transfers, the bandwidth and latency requirements trump delivery accuracy. Therefore, it is possible that some data may be lost in this pursuit. Isochronous transfers were created for things such as multimedia content that requires on-time delivery. But if the host is too busy or something interrupts the transfer, a few missing frames of video or a few milliseconds of audio are considered expendable.

Application Programmer's Interface

The FrontPanel application provides a turnkey method to make basic user interaction available to your FPGA hardware. But it is not suitable for all applications, particularly those which require further data processing on the PC side of the interface or when data transfer between the PC and FPGA is required. In these cases, a custom software application is usually a better fit. To this end, Opal Kelly provides the FrontPanel Application Programmer's Interface (API), a consistent (and in some cases cross-platform) interface to the underlying interface driver layer.

The FrontPanel API contains methods which communicate via the USB or PCIe bridge on the device, but the methods have been specifically designed to interface with FPGA hardware in a manner which is consistent with most hardware designs. The API provides methods to interface directly with the FrontPanel HDL modules such as wires, triggers, and pipes. Because of this abstraction, some flexibility in the hardware interface is sacrificed for a dramatically reduced development cycle (and learning curve!) for connecting your FPGA hardware to your custom software.

The library is written in C++ and is provided as a dynamically-linked ibrary. However, Python, Java, and C# versions of the API are also available and can make FPGA development even faster. Because the Python, Java, and C# APIs are generated automatically from the C++ API, most of the methods are identical and you can use the same API reference for all languages. Futher, the methods to communicate with PCIe and USB device are either identical or very similar.

API Reference Guide

The API documentation provided in this User's Manual gives a general overview of how the FrontPanel API is organized and used. More detailed information about the specific calling methods and parameters can be found in the API Reference Guide.

The API Reference Guide can be found online at the following URL:

http://library.opalkelly.com/library/FrontPanelAPI/index.html

The Guide is also installed with your FrontPanel software (Windows only).

Samples

Often, the best way to learn how to apply a programming interface is to see examples of its application. We encourage you to go through all of our samples to see how applications can be built with the FrontPanel SDK. If you have problems with your own design, it is a good practice to revisit our samples.

Organization

The FrontPanel API is provided as a dynamically-linked library that you include with your application. The interface to the DLL is C, but a C++ wrapper is provided to make the entire DLL appear as if it were a native C++ class in your application.

The library contains a small number of classes which you then instantiate within your code. The details of the USB or PCIe connection between the FPGA and your PC disappear within the neat confines of the API. These classes are shown in the table below in further detailed in what follows.

Class	Description
okCPLL22150	This is a container class providing methods and structure used to configure the Cypress 22150 PLL on the XEM3001 and XEM3005. An instance of this class can be created and used to program the on-board PLL or the class can be generated from the EEPROM settings.
okCPLL22393	This is a container class providing methods and structure used to configure the Cypress 22393 PLL on the XEM6010, XEM3010, and XEM3050. An instance of this class can be created and used to program the on-board PLL or the class can be generated from the EEPROM settings.
okCFrontPanel	This is the base class used to find, configure, and communicate with a FrontPanel-enabled device. The methods in the API are organized into four main groups: Device Interaction, Device Configuration, and FPGA Communication.

Loading the Library

The FrontPanel API is a dynamically-linked library that must be "loaded" into your application at runtime. To do so, you simply call `okFrontPanelDLL_LoadLib` before you use any of the API functionality:

```
// Load the FrontPanel DLL
if (FALSE == okFrontPanelDLL_LoadLib(NULL)) {
    printf("Could not load FrontPanel DLL\n");
    exit(-1);
}
```

This method takes one argument. If `NULL`, it will load the DLL from the application's current path. You can, alternatively, supply the full path and filename to the DLL to load it from a different location.

The okCFrontPanel Class

This class is the workhorse of the FrontPanel API. It's methods are organized into three main groups: Device Interaction, Device Configuration, and FPGA Communication.

In a typical application, your software will perform the following steps:

1. Create an instance of okCFrontPanel.
2. Using the Device Interaction methods, find an appropriate XEM with which to communicate and open that device.
3. Configure the device PLL (for devices with an on-board PLL).
4. Download a configuration file to the FPGA using ConfigureFPGA(...).
5. Perform any application-specific communication with the FPGA using the FPGA Communication methods.

Device Interaction (USB and PCI Express)

As much as the API encapsulates the underlying details of the hardware interface, the fact remains that the module is a USB or PCIe device and therefore must play by the rules. These methods provide a means to iterate all attached FrontPanel devices, query certain information about each one, and ultimately open a particular device for communication. These methods are summarized in the following table. For brevity, arguments have been removed. Please see the API reference manual for more details.

Method	**Description**
GetDeviceCount	Returns the number of FrontPanel devices attached to the PC. On Windows, this method counts all devices not already open. On Linux and Mac, this counts -all- devices. This method also queries information about each device which can be retrieved using the GetDeviceListXXX methods below.
GetDeviceListModel	Retrieves the board model of a connected device.
GetDeviceListSerial	Retrieves the serial number of a connected devicce.
OpenBySerial	Opens a device (with matching serial number) for communication.
GetDeviceMinorVersion	Retrieves the current firmware minor version. DEPRECATED: GetDeviceInfo replaces this method. This method will be removed in a future version of FrontPanel.
GetDeviceMajorVersion	Retrieves the current firmware major version. DEPRECATED: GetDeviceInfo replaces this method. This method will be removed in a future version of FrontPanel.
GetSerialNumber	Returns a 10-digit serial number unique to each device. This serial number may be used to select a specific device among those available. The serial number is set at the factory and is not user-modifiable. DEPRECATED: GetDeviceInfo replaces this method. This method will be removed in a future version of FrontPanel.

Device Configuration

Once an available device has been opened, these methods allow you to configure it's available features such as PLL settings and EEPROM parameters and to download configuration data to the FPGA.

Method	Description
GetDeviceID	Returns a device identification string stored in the device. Unlike the serial number, this string may be changed by the user using the API or the FrontPanel application. It is not guaranteed to be unique. DEPRECATED: GetDeviceInfo replaces this method. This method will be removed in a future version of FrontPanel.
SetDeviceID	Allows the user to set the device ID.
LoadDefaultPLLConfiguration	Configures the PLL with settings stored in EEPROM.
GetPLLxxxConfiguration	Retrieves the current on-board PLL configuration. (xxx is either 22150 or 22393)
SetPLLxxxConfiguration	Sets the on-board PLL to a given configuration. (xxx is either 22150 or 22393)
GetEepromPLLxxxConfiguration	Retrieves the PLL configuration stored in the on-board EEPROM. (xxx is either 22150 or 22393)
SetEepromPLLxxxConfiguration	Programs the on-board EEPROM with a PLL configuration for later retrieval. (xxx is either 22150 or 22393)
ConfigureFPGA	Downloads a Xilinx configuration bitfile to the FPGA.
ConfigureFPGAFromMemory	Similar to above, but with the configuration file contents in memory.
ConfigureFPGAWithReset	(USB 3.0 only) Downloads a Xilinx configuration bitfile to the FPGA and provides a reset profile to perform after configuration.
ConfigureFPGAFromMemoryWithReset	(USB 3.0 only) Similar to above, but with the configuration file contents in memory.
FlashEraseSector	(USB 3.0 only) Erases a single sector in user flash.
FlashWrite	(USB 3.0 only) Writes data to user flash.
FlashRead	(USB 3.0 only) Reads data from user flash.

FPGA Communication

Once the FPGA has been configured, communication between the application and the FPGA hardware is done through these methods. The FPGA is connected directly to the bridge device (USB microcontroller or PCIe bridge) on the XEM. These methods communicate through that connection and require that an instance of the HDL module okHostInterface be installed in the FPGA configuration.

A brief description of the API methods is in the table below. The way the API and FrontPanel HDL modules communicate is described in more detail later.

Method	Description
IsFrontPanelEnabled	Checks to see that an instance of the okHostInterface is installed in the FPGA configuration.
IsFrontPanel3Supported	Returns true if the firmware on the device supports FrontPanel-3.

Method	Description
ResetFPGA	Sends a reset signal through the host interface. This is used to reset the host interface or any endpoints. It can also be used to reset user hardware.
UpdateWireIns	Updates all wire in values (to FPGA) simultaneously with the values held internally to the API.
UpdateWireOuts	Simultaneously retrieves all wire out values (from FPGA) and stores the values internally.
UpdateTriggerOuts	Retrieves all trigger out values (from FPGA) and records which endpoints have triggered since the last query.
SetWireInValue	Sets a wire in endpoint value. Requires a subsequent call to UpdateWireIns.
GetWireOutValue	Retrieves a wire out endpoint value. Requires a previous call to UpdateWireOuts.
ActivateTriggerIn	Activates a given trigger in endpoint.
IsTriggered	Returns true if a particular trigger out endpoint has been triggered since a previous call to UpdateTriggerOuts.
WriteToPipeIn	Writes data (byte array) to a pipe in.
ReadFromPipeOut	Reads data (byte array) from a pipe out.
WriteToBlockPipeIn	Writes data to a block-throttled pipe in.
ReadFromBlockPipeOut	Reads data from a block-throttled pipe out.
WriteRegister	(USB 3.0 only) Performs a single write transaction on the Register Bridge.
ReadRegister	(USB 3.0 only) Performs a single read transaction on the Register Bridge.
WriteRegisters	(USB 3.0 only) Performs multiple write transactions on the Register Bridge.
ReadRegisters	(USB 3.0 only) Performs multiple read transactions on the Register Bridge.

Communicating with Multiple Devices

In most cases, your software will communicate with a single attached device attached. However, some applications require simultaneous communication with two or more devices. Multiple-device communication is fully supported by the driver and API but will require special consideration when initializing the communication.

Querying Attached Devices

You can call the method GetDeviceCount() to determine the number of supported devices attached to the bus before opening a specific a specific device. The GetDeviceCount() method also queries the device serial numbers and board types of all the attached devices. This information can then be accessed by calling the methods GetDeviceListSerial() and GetDeviceListModel(), respectively.

Platform-Specific Behavior

Windows, Linux, and Mac OS X behave slightly differently with regards to device enumeration using the FrontPanel API. Under Windows, if any process opens a device (OpenBySerial), that device will no longer be listed on subsequent calls to GetDeviceCount() from a different process. On the other hand, Linux and Mac OS X will allow the opened device to be enumerated. It is up to the user to assure that two processes are not communicating with the same device as this can lead to data corruption or other failures.

Connecting to a Specific Device

It is expected that you would identify a specific board using the serial number (factory-assigned and not user-mutable) or using the device ID string (user configurable via FrontPanel). A typical process for opening multiple devices would then be:

1. Create two instances (call them x and y) of the okCFrontPanel.
2. Call x.GetDeviceCount() to verify that two boards are connected and to query the serial numbers and other device information.
3. Call serX = x.GetDeviceListSerial(0) to get the first device's serial number.
4. Call serY = x.GetDeviceListSerial(1) to get the second device's serial number.
5. Call x.OpenBySerial(serX) to open the first device.
6. Call y.OpenBySerial(serY) to open the second device.

Using this procedure, you would then have two instances which point to the two devices in your system. They have also been clearly associated with the specific hardware you specified, so there is no ambiguity.

PLL Configuration

Each XEM product has a programmable PLL that can be configured through the API. In many cases, your application will require a single pre-set PLL configuration which can be stored to the on-board EEPROM (not the PLL's EEPROM). In other cases, you may want to configure the PLL from your software.

Preset PLL Configuration

With a preset PLL configuration, you setup the PLL parameters using the FrontPanel Application. You then store these parameters to the on-board EEPROM for future recall. When you startup your application (and typically before FPGA configuration), you can then configure the PLL from this stored preset using a single command:

```
xem->LoadDefaultPLLConfiguration();
```

Software PLL Configuration

You can also use the PLL classes to configure PLL parameters and then load them into the PLL. This allows dynamic PLL configuration from your own software. Software PLL configuration is a bit more complicated and requires more intimate knowledge of how the PLL parameters interoperate. Please refer to the corresponding PLL datasheet for details on the PLL parameters specific to that PLL. Then refer to the API Reference Guide for the methods available to set these parameters.

System Flash (USB 3.0)

Some FrontPanel devices have a non-volatile Flash memory attached to the USB microcontroller that is used for firmware and setting storage as well as user storage. The FrontPanel API includes methods for working with this Flash. The available storage and layout of the Flash is device-dependent. Information for each device is available in the User's Manual for that device.

API Communication

The three endpoint types (Wire, Trigger, Pipe) provide a means by which the PC and FPGA communicate. Each type is suited to a specific type of data transfer and has its own associated usage and rules.

Blocking Commands

All API methods are blocking commands. This means that the call completes before it returns to the caller (your software). Therefore, you can be assured that if you perform two consecutive API calls that update registers on the FPGA, the updates from the first call are complete prior to starting the second call.

Wires

Recall that a wire is used to communicate asynchronous signal state between the host (PC) and the target (FPGA). The okHostInterface supports up to 32 Wire In endpoints and 32 Wire Out endpoints connected to it. To save bandwidth, all Wire In or Wire Out endpoints are updated at the same time and written or read by the host in one block.

All Wire In (to FPGA) endpoints are updated by the host at the same time with the call UpdateWireIns(). Prior to this call, the application sets new Wire In values using the API method SetWireInValue(). The SetWireInValue() simply updates the wire values in a data structure internal to the API. UpdateWireIns() then transfers these values to the FPGA.

All Wire Out (from FPGA) endpoints are likewise read by the host at the same time with a call to UpdateWireOuts(). This call reads all 32 Wire Out endpoints and stores their values in an internal data structure. The specific endpoint values can then be read out using GetWireOutValue().

Note: UpdateWireIns() and UpdateWireOuts() also latch all wire endpoint data at the same time. Therefore, the data available on Wire Out endpoints are all captured synchronously (with the target interface clock). Similarly, the data availble to Wire In endpoints is all latched synchronously with the target interface clock.

Endpoint Width

Wires are 16-bits wide on USB 2.0 devices, 32-bits on USB 3.0 devices, and 32-bits on PCI Express devices. The API interface width is 32 bits. The upper bits are ignored if not supported.

Triggers

Triggers are used to communicate a singular event between the host and target. A Trigger In provides a way for the host to convey a “one-shot” on an arbitrary FPGA clock. A Trigger Out provides a way for the FPGA to signal the host with a “one-shot” or other single-event indicator.

Triggers are read and updated in a manner similar to Wires. All Trigger Ins are transferred to the FPGA at the same time and all Trigger Outs are transferred from the FPGA at the same time.

Trigger Out information is read from the FPGA using the call UpdateTriggerOuts(). Subsequent calls to IsTriggered() then return ‘true’ if the trigger has been activated since the last call to UpdateTriggerOuts().

Endpoint Width

Triggers are 16-bits wide on USB 2.0 devices, 32-bits on USB 3.0 devices, and 32-bits on PCI Express devices. The API interface width is 32 bits. The upper bits are ignored if not supported.

Pipes

Pipe communication is the synchronous communication of one or more bytes of data. In both Pipe In and Pipe Out cases, the host is the master. Therefore, the FPGA must be able to accept (or provide) data on any time. Wires, Triggers, and FIFOs can make things a little more negotiable.

When data is written by the host to a Pipe In endpoint using WriteToPipeIn(...), the device driver will packetize the data as necessary for the underlying protocol. Once the transfer has started, it will continue to completion, so the FPGA must be prepared to accept all of the data.

When data is read by the host from a Pipe Out endpoint using ReadFromPipeOut(...), the device driver will again packetize the data as necessary. The transfer will proceed from start to completion, so the FPGA must be prepared to provide data to the Pipe Out as requested.

Byte Order (USB 2.0)

Pipe data is transferred over the USB in 8-bit words but transferred to the FPGA in 16-bit words. Therefore, on the FPGA side (HDL), the Pipe interface has a 16-bit word width but on the PC side (API), the Pipe interface has an 8-bit word width.

When writing to Pipe Ins, the first byte written is transferred over the lower order bits of the data bus (7:0). The second byte written is transferred over the higher order bits of the data bus (15:8). Similarly, when reading from Pipe Outs, the lower order bits are the first byte read and the higher order bits are the second byte read.

Byte Order (USB 3.0)

Pipe data is transferred over the USB in 8-bit words but transferred to the FPGA in 32-bit words. Therefore, on the FPGA side (HDL), the Pipe interface has a 32-bit word width but on the PC side (API), the Pipe interface has an 8-bit word width.

When writing to Pipe Ins, the first byte written is transferred over the lower order bits of the data bus (7:0). The second byte written is transferred over next higher order bits of the data bus (15:8) and so on. Similarly, when reading from Pipe Outs, the lower order bits are the first byte read and the next higher order bits are the second byte read.

Byte Order (PCI Express)

The HDL pipe endpoints for PCI Express designs are 64-bits wide. The API methods are 8 bits wide for both USB and PCI Express. For PCI Express devices, pipe transfers must be in multiples of 8 bytes.

Block-Throttled Pipes

Block-Throttled Pipe communication is identical to Pipe communication with the additional specification of a block size. The FPGA sends (or receives) data in blocks sized 2, 4, 6, ..., 1024 as specified by the arguments to the call. Block sizes are restricted to 64 bytes or less when using the device at full-speed.

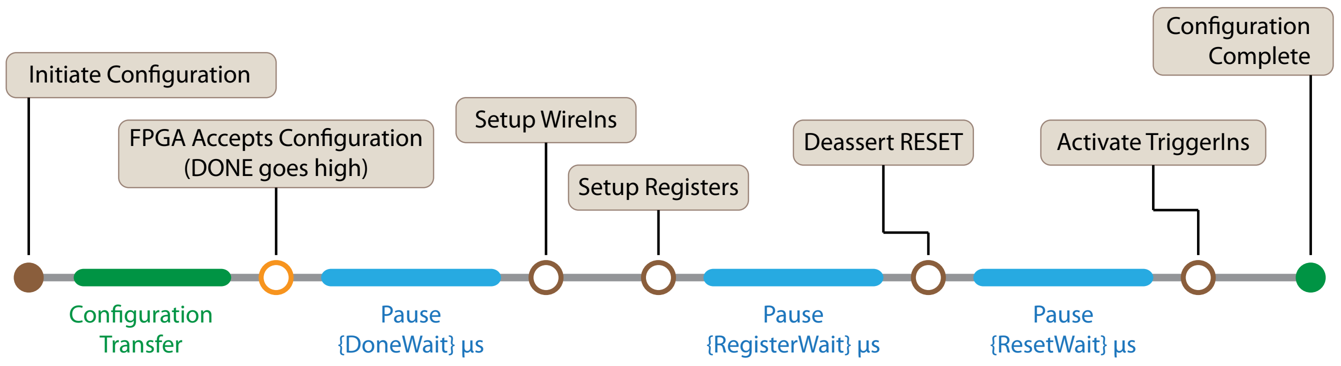
Because the FPGA has the opportunity to stall the transfer by deasserting EP_READY, the call may fail with a timeout.

Registers (USB 3.0)

The RegisterBridge HDL module provides a read / write interface with a 32-bit address range and 32-bit data width. Any address within the range may be read or written. Reads and writes through the API are performed on the hardware in the order provided by the caller and each is performed only after the previous has completed.

Reset Profiles (USB 3.0 Only)

A FrontPanel Reset Profile defines a structured approach to setting up your FPGA design after configuration has completed. The intent is to simplify initialization of your design by making sure certain inputs (such as WireIns or Registers) are setup prior to deasserting RESET. Including a Reset Profile, the FPGA configuration process includes the steps illustrated below.



1. FPGA configuration is initiated, clearing the present configuration memory.
2. RESET is asserted.
3. Configuration data is transferred to the FPGA.
4. Upon valid configuration, the FPGA indicates completion to the firmware. For Xilinx FP-GAs, this corresponds to DONE going high.
5. *Pause for {DoneWait}*, then WireIns are set to predefined values.
6. RegisterIns are set to predifined values.
7. *Pause for {RegisterWait}*, then RESET is deasserted.
8. *Pause for {ResetWait}*, then predefined TriggerIns are activated.
9. Configuration with Reset is complete.

The reset process is controlled using the Reset Profile defined with the `okTFPGAResetProfile` structure. FrontPanel API methods that accept a Reset Profile are described in the following table.

Method	Description
ConfigureFPGAWithReset	Configure the FPGA from a file and perform a Reset Profile after configura-tion.
ConfigureFPGAFromMemory-WithReset	Configure the FPGA from memory and perform a Reset Profile after configu-ration

Method	Description
GetFPGABootResetProfile	Retrieves the Reset Profile stored in Flash memory that is used for power-on configuration of the FPGA.
GetFPGAJTAGResetProfile	Retrieves the Reset Profile stored in Flash memory that is used for JTAG configuration of the FPGA.
SetFPGABootResetProfile	Sets the Reset Profile stored in Flash memory that is to be used for power-on configuration of the FPGA.
SetFPGAJTAGResetProfile	Sets the Reset Profile stored in Flash memory that is to be used for JTAG configuration of the FPGA.

```
typedef struct okFPGAResetRegisterEntry {
    UINT32   address;
    UINT32   data;
} okTFPGAResetRegisterEntry;

typedef struct okFPGAResetTriggerEntry {
    UINT32   address;
    UINT32   mask;
} okTFPGAResetTriggerEntry;

typedef struct okFPGAResetProfile {
    UINT32                     magic;
    UINT32                     configFileLocation;
    UINT32                     configFileLength;
    UINT32                     doneWaitUS;
    UINT32                     resetWaitUS;
    UINT32                     registerWaitUS;
    UINT32                     padBytes1[28];
    UINT32                     wireInValues[32];
    UINT32                     registerEntryCount;
    okTFPGAResetRegisterEntry  registerEntries[256];
    UINT32                     triggerEntryCount;
    okTFPGAResetTriggerEntry   triggerEntries[32];
    UINT8                      padBytes2[1520];
} okTFPGAResetProfile;
```

Device Sensors (USB 3.0 Only)

The Device Sensors API provides programmer access to sensing capabilities built into some Opal Kelly integration modules. These sending capabilities provide real-time measurement of select device voltages, currents, and temperatures for monitoring purposes. Device Sensors are device-specific and are listed in the corresponding device User's Manual if this feature is available.

Important portions of the API are shown in the following snippet.

```
// Query all device sensors and retrieve the collection.
ErrorCode okCFrontPanel::GetDeviceSensors(okCDeviceSensors& sensors) const;

// Get the number of sensors.
int okCDeviceSensors::GetSensorCount() const;

// Get a particular sensor.
okTDeviceSensor okCDeviceSensors::GetSensor(int index) const;

// Device Sensor structure.
typedef struct okDeviceSensor {
    int                     id;
    okEDeviceSensorType     type;
    char                    name[OK_MAX_DEVICE_SENSOR_NAME_LENGTH];
    char                    description[OK_MAX_DEVICE_SENSOR_DESCRIPTION_LENGTH];
    double                  min;
    double                  max;
    double                  step;
    double                  value;
} okTDeviceSensor;
```

GetDeviceSensors API

The device firmware manages an internal store of all device sensors and queries them periodically. The values of this store are transferred to the host using this API. Note that the update period is not programmable or known to the API but is approximately once every second or so. The sensor collection is loaded into an instance of `okCDeviceSensors` which is returned by this API. Accessor methods of this class then provide the count and query of each `okTDeviceSensor`.

The typical use case for Device Sensors is as follows:

1. Call okCFrontPanel::GetDeviceSensors to retrieve the sensor collection.
2. Call okCDeviceSensors::GetSensorCount to retrieve the collection count.
3. Iterate through the sensor count, calling GetSensor to print sensor values or search for a particular sensor name.

Device Sensor Parameters

Each sensor has several parameters that may be queried via the API. During each read of the Device Sensors, these parameters are placed into a structure `okTDeviceSensor`.

- `id` - Sensor ID (reserved for future use).
- `type` - Sensor type from the table below.
- `name` - String name of the sensor.
- `description` - Brief string describing the sensor measurement.
- `min` - Minimum measurement value.
- `max` - Maximum measurement value.
- `step` - Value step size attributable to the measurement resolution.
- `value` - Value of the measurement.

okEDeviceSensorType	Description
`okDEVICESENSOR_INVALID`	Invalid sensor
`okDEVICESENSOR_BOOL`	Boolean (0=false, 1=true)
`okDEVICESENSOR_INTEGER`	Integer value
`okDEVICESENSOR_FLOAT`	Floating point value
`okDEVICESENSOR_VOLTAGE`	Voltage with corresponding minimum, maximum, and step size
`okDEVICESENSOR_CURRENT`	Current with corresponding minimum, maximum, and step size
`okDEVICESENSOR_TEMPERATURE`	Temperature with corresponding minimum, maximum, and step size
`okDEVICESENSOR_FAN_RPM`	Fan RPM (revolutions per minute)

Device Settings (USB 3.0 Only)

The Device Settings API provides programmer access to persistent and non-persistent key / value pairs managed by the device firmware. Available Device Settings are device-specific and supported settings are listed in the corresponding device User's Manual.

Important portions of the API are shown in the following snippet.

```
// Retrieve the Device Settings module for the device.
ErrorCode okCFrontPanel::GetDeviceSettings(okCDeviceSensors& sensors);

// Get a list of setting keys.
int okCDeviceSettings::List(std::vector<str::string>& keys);

// Get / Set an integer setting.
okCDeviceSetting::GetInt(const std::string& key, UINT32 *value);
okCDeviceSetting::SetInt(const std::string& key, UINT32 value);

// Get / Set an string setting.
okCDeviceSetting::GetString(const std::string& key, std::string *value);
okCDeviceSetting::SetString(const std::string& key, std::string value);

// Delete a setting
okCDeviceSetting::Delete(const std::string& key);

// Save settings to persistent storage.
okCDeviceSetting::Save();
```

Persistent Settings

Persistent settings are stored in non-volatile memory (NVM) on the device. These settings are used to control startup behavior as well as runtime operation. Because they are stored in NVM, the setting persists when the device is powered off.

There is a separate API (`okCDeviceSetting::Save`) which must be called to commit settings to NVM. Settings are only changed in volatile memory until this method is called.

Non-Persistent Settings

Non-persistent settings are runtime settings that are not necessarily stored in NVM. These settings usually control or monitor runtime operation. For example, controlling an on-device fan would be a runtime setting and changes to that setting affect device behavior immediately.

Note that a particular setting may be both Persistent and Non-Persistent. The determination is device-specific and the implication of a setting that is both Persistent and Non-Persistent is that changing the setting affects device behavior immediately and is also used during device startup to affect behavior at that time. For example, a fan setting that is both persistent and non-persistent will modify the fan operation immediately and, if written to NVM, the device will restore that operation on boot.

Setting Store

The setting store is a block of NVM in System Flash that has been set aside for persistent setting storage. This block is read at device startup and is written upon calling the API method `okCDeviceSetting::Save`.

FrontPanel API Example Usage

Below is a short code snippet that illustrates how the API might be used in a C++ application. More useful and detailed examples can be found in the **Samples** folder of the FrontPanel installation.

```
if (FALSE == okFrontPanelDLL_LoadLib(NULL)) {
    printf("Could not load FrontPanel DLL\n");
    exit(-1);
}

okCFrontPanel *dev = new okCFrontPanel();
dev->OpenBySerial();
dev->LoadDefaultPLLConfiguration();
dev->ConfigureFPGA("mybitfile.bit");

// Set a value on WireIn endpoint 0x00.
dev->SetWireInValue(0x00, 0x37);
dev->UpdateWireIns();

// Activate TriggerIn 0x40:0 (clears address pointers).
dev->ActivateTriggerIn(0x40, 0);

// Send 1024 bytes to PipeIn 0x80.
dev->WriteToPipeIn(0x80, 1024, buf);
// Read 1024 from PipeOut 0xA0.
dev->ReadFromPipeOut(0xA0, 1024, buf);

// Read the result from WireOut endpoint 0x20.
dev->UpdateWireOuts();
result = dev->GetWireOutValue(0x20);
```

Regarding Device Ownership (Multithread or Multiprocess Access)

In general, once an instance of okCFrontPanel has been opened, that instance "owns" the device. That means that, while the API will allow you to create another instance and communicate with the same device, there are likely going to be problems with doing so.

In situations where you must have multiple threads or processes communicating with the same device, it is better to have a single owner of the device instance and route all calls through that owner.

The exception to this is GetDeviceCount() and the associated calls under Linux and Mac OS X. (This exception does not apply to Windows.) You can call this method at any time (even before opening a device) to determine the number of attached FrontPanel devices and retrieve their model numbers, and serial numbers. You may not retrieve the Device ID string without opening the device and that implies "owning" the device.

32-bit and 64-bit Architectures

The FrontPanel API is distributed for 32-bit and 64-bit architectures on Windows and Linux. If you have a 32-bit version of Windows installed, you will only need the 32-bit API since Windows/32 cannot run applications build for 64-bit architectures.

If you have a 64-bit version of Windows installed, the OS can run both 32-bit and 64-bit applications. The FrontPanel API you choose should match the architecture of the application you are using. Typically, 32-bit applications are installed in "`Program Files (x86)`" and 64-bit applications are installed in "`Program Files`".

	Windows (32-bit)	**Windows (64-bit)**
32-bit Application	32-bit API	32-bit API
64-bit Application	-	64-bit API

Windows DLL Usage

The Microsoft Visual C++ 2013 Redistributable is required to use the FrontPanel DLL and the wrapped APIs because the DLLs are compiled against Microsoft libraries. Your software installation package should install the appropriate redistributable (32-bit and 64-bit versions are available). The redistributable package is available for free from Microsoft's website.

Note that the redistributable for the corresponding architecture of the DLL needs to be installed, regardless of the Windows OS architecture. Therefore, if you're using the 32-bit version of Python on a 64-bit version of the Windows OS, you will need to have the 32-bit version of the redistributable installed.

http://www.microsoft.com/en-us/download/details.aspx?id=40784

Wrapped APIs

The FrontPanel API is available in C#, Java, Ruby, and Python. These APIs are similar to the native (C/C++) API, but have a few peculiarities. The best place to start is our DESTester sample that is available on in all of the wrapped APIs.

Getters and Setters

Getters and Setters are the methods used to access member variables in a structure (such as okTDeviceInfo). Differences exist depending on the specific language conventions. Below are some examples of the getters and setters. In particular, the Java API uses functions to access member variables that are prefixed with "get" or "set".

```
// C/C++
printf(devInfo.serialNumber);
regEntry.address = 0x00001234;

// C#
System.Console.Write(devInfo.serialNumber);
regEntry.address = 0x00001234;

// Java
System.out.println(devInfo.getSerialNumber());
regEntry.setAddress = 0x00001234;

# Python
print(devInfo.serialNumber)
regEntry.address = 0x00001234

# Ruby
puts devInfo.serialNumber
regEntry.address = 0x00001234
```

Data Types

Whenever possible, the corresponding data types in the language have been mapped transparently to the native API. There are a few exceptions to this that should be noted, however.

Java : Long Instead of Unsigned Int

The Java language does not have unsigned types. Therefore, when using the 32-bit unsigned int for methods such as GetWireOutValue and SetWireInValue, it is best to use Java longs. Otherwise, the most significant bit will map and sign-extend incorrectly.

```
// Java
device.SetWireInValue(0x00, 0x876543210L);      // Use long constants
0x87654321L == device.GetWireOutValue(0x20);    // Expect results as longs
```

Byte Arrays

Byte arrays are used when bulk data is handed to or received from the API. Examples include the Pipe transfer APIs and the Flash memory read/write APIs. Byte arrays are handled differently in different languages.

```
// C/C++
unsigned char *data = new unsigned char[deviceInfo.flashSystem.sectorSize];
device.FlashWrite(address, length, data);

// C#
byte[] data = new byte[(int)deviceInfo.flashSystem.sectorSize];
device.FlashWrite(address, length, data);

// Java
byte[] data = new byte[(int)deviceInfo.getFlashSystem().getSectorSize()];
device.FlashWrite(address, length, data);

# Python
data = bytearray(deviceInfo.flashSystem.sectorSize)
device.FlashWrite(data)
```

Python API

The Python API is built as an import library to be used with the Python interpreted programming language. Python is a powerful extensible language with a clear syntax, making it ripe for the FrontPanel API add-on. The Python API is built using the C++ API as a foundation, so the similarities are pervasive.

The Python API is compiled for each supported platform into a shared object file (DLL under Windows or .so under Linux) and distributed along with a couple Python files that define the package. The Python interpreter can access the API methods through this shared object and Python package.

Required Files

The Python API distribution includes four files as listed below:

- `__init__.py`
- `__version__.py`
- `ok.py`
- `_ok.pyd` (Windows, architecture-specific for 32-bit/64-bit)
- `_ok.so` (Linux, Mac OS X, architecture-specific)

These four files need to be in the current working directory where Python is started. Alternatively, they may be added to the Python **site-packages** directory within your Python distribution. Refer to the Python manual to see how this is done.

Example Usage

Using the API from Python is quite easy and can be done scripted or interactively. Below is an example interaction with the Counters sample project.

```
>>> import ok
>>> dev = ok.okCFrontPanel()
>>> pll = ok.okCPLL22150()
>>> dev.GetEepromPLL22150Configuration(pll)
1
>>> dev.SetPLL22150Configuration(pll)
1
>>> pll.GetOutputFrequency(0)
100.0
>>> dev.ConfigureFPGA('c:\counters.bit')
1
>>> dev.ActivateTriggerIn(0x40,0)
1
>>>
```

Java API

The Java API is built as an extension library to be used with Sun's compiled Java language. It it built on top of the JNI (Java Native Interface). The API is distributed as a shared library and a Java archive (JAR file).

Required Files

There are only two required files for the Java API: the shared library and the Java archive:

- `okjFrontPanel.dll` (Windows, architecture-specific for 32-bit/64-bit)
- `okjFrontPanel.so` (Linux, architecture-specific for 32-bit/64-bit)
- `libokjFrontPanel.jnilib` (Mac OS X)
- `okjFrontPanel.jar`

Under Windows, you can keep the DLL in the directory where you run **java**. Under Linux, the shared object should be placed within your java.class.path. For example, under the SuSE 9.2 Linux distribution, you would copy the file to: **/usr/lib/jre/lib/i386**.

Example Usage

Within a Java source that uses the Java FrontPanel API, you need to import the FrontPanel classes using the following line:

```
import com.opalkelly.frontpanel.*;
```

To actually load the FrontPanel library into Java, you will also need to make the following System call before using any FrontPanel API objects:

```
System.loadLibrary("okjFrontPanel");
```

Compiling a Java application for use with the Java API can be done on the command line using **javac** with the **-classpath** argument to specify the Java API JAR as shown below.

```
javac -classpath okjFrontPanel.jar MyClass.java
```

Likewise, when running the application, you need to add the Java API JAR to the classpath:

```
java -classpath .;okjFrontPanel.jar MyClass    # Windows
java -classpath .:okjFrontPanel.jar MyClass    # Mac & Linux
```

A thorough example of the Java API is included in the DESTester application. Shown below is the Python example above transformed into Java.

```
import com.opalkelly.frontpanel.*;
public class JavaAPITest {
    public void TestMethod() {
        dev = new okCFrontPanel();
        pll = new okCPLL22150();
        dev.GetEepromPLL22150Configuration(pll);
        dev.SetPLL22150Configuration(pll);
        System.out.println("PLL Output: " + pll.GetOutputFrequency(0) + " MHz");

        dev.ConfigureFPGA("c:/counters.bit");
        dev.ActivateTriggerIn((short)0x40, (short)0);
    }
}
```

C# API

The C# API is built as an extension library using the Platform Invoke (PInvoke) capability of .NET.

Required Files

There are only two required files for the C# API: the shared library and the C# library:

- `libFrontPanel-pinv.dll` - Shared library, architecture-specific for 32-bit/64-bit
- `libFrontPanel-csharp.dll` - C# library

The C# library must be added to your C# project as a Reference. The shared library must be placed in the runtime directory of your application.

Example Usage

Within your C# source, you will need to specify the namespace for the FrontPanel API:

```
using OpalKelly.FrontPanel;
```

At runtime, the FrontPanel API classes will automatically load the shared library as necessary.

A good example of the C# API is included in the DESTester application. Shown below is a brief code snippet:

```
using OpalKelly.FrontPanel;
class CsharpAPITest {
    public void TestMethod() {
        dev = new okCFrontPanel();
        dev.OpenBySerial(““);
        dev.LoadDefaultPLLConfiguration();
        dev.ConfigureFPGA(“c:/counters.bit”);
        dev.ActivateTriggerIn((short)0x40, (short)0);
    }
}
```

FrontPanel DLL

On the Windows platform, a dynamically-linked library (DLL) is available. On other platforms (Linux, Mac OS X, and QNX), this library is known as a shared-object. We will use the terms DLL and shared-object to mean the same. This DLL makes it possible to call the FrontPanel API from other programming languages (such as C, C++, VisualBasic) as well as from many third-party software applications such as LabVIEW and Matlab. It also means that you don't need to have a precompiled API library specific to your compiler.

The FrontPanel DLL is provided as three files listed in the table below:

Filename	Description

`okFrontPanel.dll`
`libokFrontPanel.so`
`libokFrontPanel.dylib`
	The FrontPanel DLL binary (Windows, Linux, Mac, respectively). This file needs to be located with your application executable or, for third-party software, in the appropriate DLL location.

`okFrontPanel.lib`	Referred to as an "import library," this file contains references necessary to call the functions in the DLL. This is typically required only for C and C++ applications and is statically linked to your application at compile time.
`okFrontPanelDLL.h`	This header file contains the FrontPanel DLL entrypoints corresponding to the functions in the import library.

In most cases, each class method has a corresponding DLL entrypoint. This makes it easy to refer to the standard API documentation for calling information. One notable difference is that most DLL entrypoints require a pointer argument. This pointer is actually the pointer to the allocated C++ class object. Note, however, that this object is allocated and deallocated using DLL entrypoints and therefore the *DLL does NOT require C++* and can be used in any C application.

QNX Usage

QNX is a real-time operating system that runs on a number of architectures, including Intel x86. From a build environment perspective, it is very similar to Linux and other UNIX-like operating systems. Our samples should build and run under QNX without modification. The typical application startup procedure is to create a new QNX project from within the Momentix IDE using QNX templates, then copy the source files from one of the samples into that project. You will also need to include the shared object file with your compiled binary.

Under the build settings (Makefile), you will need to change the `LDFLAGS` setting to include the QNX USB library as shown below.

```
LDFLAGS+=-lang-c++ -lusbdi
```

Memory Allocation

Due to the way the QNX USB stack operates, you must allocate memory differently if it is going to be used for any FrontPanel Pipe transfers. See the code example below as reference. Note that this only applies to buffers used for pipe transfers.

```
#include <sys/usbdi.h>

pBuffer = new unsigned char[65536];             // Typical C++ allocation
pBuffer = (unsigned char *)usbd_alloc(65536);   // QNX pipe buffer allocation

delete [] pBuffer;    // Typical C++ deallocation of a buffer
usbd_free(pBuffer);   // QNX deallocation of pipe buffers
```

Example Usage (C/C++)

When using the DLL in a compiled C/C++ application, you will need to link the `okFrontPanel.lib` file with your application. This file contains references necessary to call the functions in the DLL. You will also need to include the file `okFrontPanelDLL.h` in each source file that calls the DLL.

Initialization

Before calling functions within the DLL, you need to load the DLL library. This is done with the following call:

```
// Initialize the DLL and load all entrypoints.
if (FALSE == okFrontPanelDLL_LoadLib()) {
    printf("ERROR: FrontPanel DLL could not be initialized.\n");
}
```

There are a few reasons this call (LoadLib) may fail:

- The DLL itself must be where the software thinks it is. If you do not pass a full path to the function, it will assume it is in the present working directory of the application. Hint: This is not necessarily the location of the executable!

- The DLL must be the same architecture (32-bit or 64-bit) as the application. If you're building a 64-bit application, use the 64-bit DLL. If you're building a 32-bit application, use the 32-bit DLL.
- The DLL requires the Microsoft Visual C++ 2010 Redistributable to be installed on the target machine. Be sure to install the redistributable that matches the architecture of the DLL.

Constructing and Destructing Objects

The FrontPanel API is an object-oriented library but the DLL is strictly C-style. Therefore, methods have been provided in the DLL for creating and destroying the objects such as okCPLL22150 and okCFrontPanel. An object must be created before its methods can be called. An object should also be destructed when you are done using it.

```
okFRONTPANEL_HANDLE dev;
dev = okFrontPanel_Construct();
...
    // Use the 'dev' object.
...
okFrontPanel_Destruct(dev);
```

Calling Methods

Each DLL method that acts on an object has an additional required argument that indicates which object is being acted upon. In C++, this additional argument is implied by the object-oriented nature of the language. In the DLL this argument must be explicitly provided.

C++ Wrapper

Also included in the `okFrontPanel.h` file is a C++ wrapper for the DLL. This provides a full C++ object class so that you do not have to call the C-style DLL methods from your C++ application. Most of the samples are written using this C++ wrapper.

```
okFRONTPANEL_HANDLE dev;
okPLL22150_HANDLE pll;

// Construct device and PLL objects.
dev = okFrontPanel_Construct();
pll = okPLL22150_Construct();

// Setup the PLL.
okPLL22150_SetVCOParameters(pll, 400, 48);
okPLL22150_SetDiv1(pll, DivSrc_VCO, 8);
okPLL22150_SetOutputSource(pll, 0, ClkSrc_Div1ByN);
okPLL22150_SetOutputEnable(pll, 0, true);

// Configure the XEM PLL.
okFrontPanel_OpenBySerial(dev, NULL);
okFrontPanel_SetPLLConfiguration(dev, pll);

// Finished with the PLL.
okPLL22150_Destruct(pll);

...
    // Use the ‘dev’ object.
...


okFrontPanel_Destruct(dev);
```

Example Usage (Matlab)

Matlab provides a convenient way to extend its own capabilities by calling user-provided DLL functions. This is done using a few native Matlab calls: loadlibrary, calllib, libisloaded, libfunctions, libfunctionsview.

For example, to load the FrontPanel DLL into Matlab for use, the following syntax can be used:

```
if ~libisloaded(‘okFrontPanel’)
    loadlibrary(‘okFrontPanel’, ‘okFrontPanelDLL.h’);
end
```

You can view the calling conventions and conversions Matlab has applied to the DLL methods by calling the command “libfunctionsview(‘okFrontPanel’)”. An example way to call the DLL:

```
% Create a device structure:
xid.ptr = 0;
xid.serial = ‘‘;
xid.deviceID = ‘‘;
xid.major = 0;
xid.minor = 0;

% Construct an XEM3001v2 and open the first device:
xid.ptr = calllib(‘okFrontPanel’, ‘okFrontPanel_Construct’);
[ret, x] = calllib(‘okFrontPanel’, ‘okFrontPanel_Open’, xid.ptr, 0);
[xid.major, x] = calllib(‘okFrontPanel’, ...
        ‘okFrontPanel_GetDeviceMajorVersion’, xid.ptr);
[xid.minor, x] = calllib(‘okFrontPanel’, ...
        ‘okFrontPanel_GetDeviceMinorVersion’, xid.ptr);
[x, xid.serial] = calllib(‘okFrontPanel’, ...
        ‘okFrontPanel_GetSerialNumber’, xid.ptr, ‘           ‘);
[x, xid.deviceID] = calllib(‘okFrontPanel’, ...
        ‘okFrontPanel_GetDeviceID’, xid.ptr,
        ‘                                  ‘);
```

Matlab API

While the above example shows how to use the FrontPanel DLL from within Matlab, we have already provided a more thorough version of this API for your usage. It is provided as a fully-functioning sample of the DLL usage from within Matlab and utilizes Matlab's object-oriented structure to provide an API that is very similar to the C++ API in usage.

DLL Header File

Due to a bug in Matlab's DLL usage, a slightly modified DLL header file must be used when accessing the API through Matlab. This revised header defines the HANDLE objects as `unsigned long` rather than `void *`. If the revised header file is not used, memory leaks will occur in Matlab.

Support Status

Please note that the Matlab API is not officially supported by Opal Kelly. While it is not officially supported, we would like to keep it up-to-date. Please contact us via email if you have any suggested changes to the Matlab API.

HDL Modules

The use of FrontPanel components to control and observe pieces of your FPGA design requires the instantiation of one or more modules in your toplevel HDL. These modules can quickly and easily be added into an existing or new design and take care of all the dirty work of communicating with the FrontPanel software.

The host interface is the block which connects directly to pins on the FPGA which are connected to the USB microcontroller. This is the entry point for FrontPanel into your design.

The endpoints connect to a shared control bus on the host interface. This internal bus is used to shuttle the endpoint connections to and from the host interface. Several endpoints may be connected to this shared bus. FrontPanel uses endpoint addresses to select which endpoint it is communicating with, so each endpoint must have its own unique address to work properly.

Endpoint Types

FrontPanel supports three basic types of endpoints: Wire, Trigger, and Pipe. Each can either be an input (from host to target) or output (from target to host). Each endpoint type has a certain address range which must be used for proper operation. The address is specified at the instantiation of the endpoint module in your design.

Endpoint Type	Address Range	Sync/Async	Data Type
Wire In	0x00 - 0x1F	Asynchronous	Signal state
Wire Out	0x20 - 0x3F	Asynchronous	Signal state
Trigger In	0x40 - 0x5F	Synchronous	One-shot
Trigger Out	0x60 - 0x7F	Synchronous	One-shot

Endpoint Type	Address Range	Sync/Async	Data Type
Pipe In	0x80 - 0x9F	Synchronous	Multi-byte transfer
Pipe Out	0xA0 - 0xBF	Synchronous	Multi-byte transfer

Endpoints are instantiated in your HDL design and connected to the okHost target ports. Each endpoint also has one or more ports which are connected to various signals in your design, depending on the endpoint module.

Endpoint Addresses

Endpoints attach to the host interface on a shared bus. To properly route signals between the host (PC) and target endpoints, each endpoint must be assigned a unique 8-bit address. For performance reasons (to minimize USB transactions), each endpoint type has been assigned an address range as indicated in the table above. When assigning addresses to your endpoints, be sure to follow these ranges.

The endpoint address is assigned in HDL through an additional 8-bit input port on the endpoint instance. Example instantiation for each endpoint type are shown in the sections below.

Register Bridge (USB 3.0 Only)

FrontPanel for USB 3.0 devices supports Wires, Triggers, and Pipes, as well as an additional Register Bridge endpoint. This register bridge provides a 32-bit address space of 32-bit data words for a total of 16 GB. User HDL responds to read and write requests from this Register Bridge, effectively creating a large synchronous register file.

Endpoint Data Widths

Endpoint data widths vary depending on the interface according to the table below.

Endpoint Type	USB 2.0	USB 3.0	PCI Express
Wire	16	32	32
Trigger	16	32	32
Pipe	16	32	64
Register Bridge - Address	-	32	-
Register Bridge - Data	-	32	-

Host Interface Clock Speed

The HDL host interface is a slave interface from the host. It runs at a fixed clock rate that is dependent upon the interface type for the device.

- USB 2.0 interfaces run at 48 MHz (20.83 ns clock period)
- USB 3.0 interfaces run at 100.8 MHz (9.92 ns clock period)
- PCI Express (x1) interfaces run at 50 MHz (20 ns clock period)

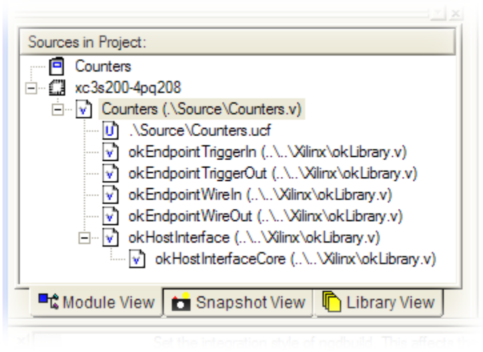
Building FPGA Projects with FrontPanel HDL Modules

The FrontPanel HDL Modules are provided as pre-synthesized files which get included in your design flow. The following table lists these files and describes its content. By default, these files are installed at **C:\Program Files\Opal Kelly\FrontPanelUSB\FrontPanelHDL**. In this directory are several subdirectories that contain HDL modules built for different different devices. Choose the library and NGC files suitable for your device.

Filename	Description
`okLibrary.v`	Verilog file containing black-box modules for Verilog projects.
`okLibrary.vhd`	VHDL file containing black-box modules for VHDL projects.
`okCore.ngc`	Pre-synthesized Xilinx module for the Host.
`okWireIn.ngc`	Pre-synthesized Xilinx module for the Wire In endpoint.
`okWireOut.ngc`	Pre-synthesized Xilinx module for the Wire Out endpoint.
`okTriggerIn.ngc`	Pre-synthesized Xilinx module for the Trigger In endpoint.
`okTriggerOut.ngc`	Pre-synthesized Xilinx module for the Trigger Out endpoint.
`okPipeIn.ngc`	Pre-synthesized Xilinx module for the Pipe In endpoint.
`okPipeOut.ngc`	Pre-synthesized Xilinx module for the Pipe Out endpoint.
`okBTPipeIn.ngc`	Pre-synthesized Xilinx module for the Block-Throttled Pipe In endpoint.
`okBTPipeOut.ngc`	Pre-synthesized Xilinx module for the Block-Throttled Pipe Out endpoint.

The Host Interface is comprised of two components - a core component which is pre-synthesized and a wrapper component (in `okLibrary.v` or `okLibrary.vhd`) which includes the core component as well as IOBs required for the connections to FPGA pins.

When you start a new design, you should copy `okLibrary.v` or `okLibrary.vhd` into the directory with your other sources and add them to your project. This file will be synthesized just like your other modules except that the HDL is mostly just a placeholder for the modules that have been pre-synthesized. When properly added to a project, Project Navigator will list the source follows similar to what is shown below:


```
Sources in Project:
  Counters
  xc3s200-4pq208
    Counters (.\Source\Counters.v)
      .\Source\Counters.ucf
      okEndpointTriggerIn (..\..\Xilinx\okLibrary.v)
      okEndpointTriggerOut (..\..\Xilinx\okLibrary.v)
      okEndpointWireIn (..\..\Xilinx\okLibrary.v)
      okEndpointWireOut (..\..\Xilinx\okLibrary.v)
      okHostInterface (..\..\Xilinx\okLibrary.v)
        okHostInterfaceCore (..\..\Xilinx\okLibrary.v)
```

You should also copy the pre-synthesized files (`*.ngc`) that you use into your project directory. You do not need to copy module files that you are not using. The `.ngc` files will then be used by the Xilinx tools during the Translate step in order to completely build the FPGA configuration file.

HDL Modules - USB 2.0

XEM3001v1 Note

The first PCB revision of the XEM3001 (date code: 20040301) had an 8-bit host interface. All newer implementations have a 16-bit interface. For the purposes of this section, the only things which change are the HI_DATA, TI_DATA busses, as well as the widths of the endpoint connections (such as EP_DATAIN and EP_DATAOUT). For the XEM3001v1, simply substitute an 8-bit bus in those places.

FPGA Resource Requirements

The FrontPanel-enabling modules have been designed to consume as few resources as possible within the FPGA. The resource requirements for each block are listed in the tables below. Keep in mind that these are requirements for an endpoint with all bits used. In many cases, the place and route tools will optimize and remove unused components.

Resource	Slice FFs	4-in LUTs	Block RAMs
Host Interface	33	49	0
Wire In	16	14	0
Wire Out	8	5	0
Trigger In	32	21	0
Trigger Out	27	15	0
Pipe In	9	10	0
Pipe Out	0	6	0
BT Pipe In	?	?	?

Resource	Slice FFs	4-in LUTs	Block RAMs
BT Pipe Out	?	?	?

Wire-OR

Multiple endpoints are attached to the `ok2` bus on the `okHost` by using a Wire-OR. Each endpoint is told when it can assert its data on the bus. At all other times, it drives 0. The Wire-OR component performs a bitwise OR operation on each bit of the bus and outputs the result. In this manner, multiple endpoints can share a bus without requiring the use of tristates or a large mux.

The okWireOR is provided as a parameterized helper module in `okLibrary.v` and `okLibrary.vhd`. Please refer to the provided samples to see how to instantiate this module.

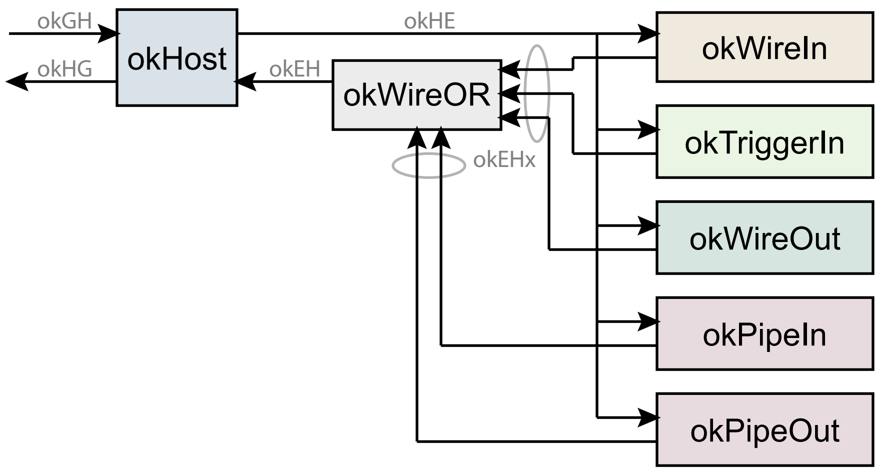
The Host Interface

The host interface is the gateway for FrontPanel to control and observe your design. It contains relatively simple logic that lets the USB microcontroller on the device communicate with the various endpoints within the design. Exactly one host interface must be instantiated in any design which uses the FrontPanel components.

The Host Interface component is the only block which is synthesized with your design. It contains a Host Interface core component (provided as a pre-synthesized module) as well as the necessary IOB components to connect to the host interface pins of the FPGA.

NOTE: The okHost is contained in `okLibrary.v` or `okLibrary.vhd`. Some details change from device to device so the exact pinouts may differ slightly from the documentation below. Please see the Opal Kelly samples in the FrontPanel installation directory for examples specific to each supported device.

The diagram below illustrates the structural relationships between the various endpoints, the okWireOR, and okHost modules.



okHost

This module must be instantiated in any design that makes use of FrontPanel virtual interface components. The following signals need to be connected directly to pins on the FPGA which go to the USB microcontroller on the device. For a listing of the pin locations for a particular product, please see the user's manual for that device.

Signal	Direction	Description
`HI_IN[7:0]`	Input	Host interface input signals.
`HI_OUT[1:0]`	Output	Host interface output signals.
`HI_INOUT[15:0]`	In/Out	Host interface bidirectional signals.

The remaining ports of the okHost are connected to a shared bus inside your design. These signals are collectively referred to as the target interface bus. Each endpoint must connect to these signals for proper operation.

Signal	Direction	Description
`OK1[30:0]`	Output	Control signals to the target endpoints.
`OK2[16:0]`	Input	Control signals from the target endpoints.
`TI_CLK`	Output	Buffered copy of the host interface clock (48 MHz). This signal does not need to be connected to the target endpoints because it is replicated within OK1.

Instantiation of the okHost is simple in either VHDL or Verilog. Use the templates below in your toplevel HDL design. A more detailed listing can be found later in this manual as one of the examples.

Verilog Instantiation:

```
okHost hostIF (.hi_in(hi_in),
    .hi_out(hi_out), .hi_inout(hi_inout),
    .ti_clk(ticlk), .ok1(ok1), .ok2(ok2));
```

VHDL Instantiation:

```
okHI : okHost port map (hi_in => hi_in,
    hi_out => hi_out, hi_inout => hi_inout,
    ti_clk => ticlk, ok1 => ok1, ok2 => ok2);
```

Each endpoint is connected to 48 target interface pins on the okHost module. The direction is from the perspective of the endpoint module.

Signal	Direction	Description
`OK1[30:0]`	Input	Interface control (HI to target).
`OK2[16:0]`	Output	Interface control (Target to HI).

These signals are present in every endpoint. In the signal tables for the independent endpoints below, we have left out these common signals.

okWireIn

In addition to the target interface pins, the okWireIn adds a single 16-bit output bus called `EP_DATAOUT[15:0]`. The pins of this bus are connected to your design as wires and act as asynchronous connections from FrontPanel components to your HDL.

When FrontPanel updates the Wire Ins, it writes new values to the wires, then updates them all at the same time. Therefore, although the wires are asynchronous endpoints, they are all updated at the same time *on the host interface clock*.

Signal	Direction	Description
`EP_DATAOUT[15:0]`	Output	Wire values output. (sent from host)

Verilog Instantiation:

```
okWireIn wire03 (.ok1(ok1),
    .ep_addr(8'h03), .ep_dataout(ep03data));
```

VHDL Instantiation:

```
wire03 : okWireIn port map (ok1 => ok1,
    ep_addr => x"03", ep_dataout => ep03data);
```

okWireOut

An okWireOut module adds a single 16-bit input bus called `EP_DATAIN[15:0]`. Signals on these pins are read whenever FrontPanel updates the state of its wire values. In fact, all wires are captured simultaneously (synchronous to the host interface clock) and read out sequentially.

Signal	Direction	Description
`EP_DATAIN[15:0]`	Input	Wire values input. (to be sent to host)

Verilog Instantiation:

```
okWireOut wire21 (.ok1(ok1), .ok2(ok2),
    .ep_addr(8'h21), .ep_datain(ep21data));
```

VHDL Instantiation:

```
wire21 : okWireOut port map (ok1 => ok1, ok2 => ok2,
    ep_addr => x"21", ep_datain => ep21data);
```

okTriggerIn

The okTriggerIn provides `EP_CLK` and `EP_TRIGGER[15:0]` as interface signals. The Trigger In endpoint produces a single-cycle trigger pulse on any of `EP_TRIGGER[15:0]` which is synchronized to the clock signal `EP_CLK`. Therefore, the single-cycle does not necessarily have to be a single host interface cycle. Rather, the module takes care of crossing the clock boundary properly.

Signal	Direction	Description
`EP_ADDR[7:0]`	Input	Endpoint address.
`EP_CLK`	Input	Clock to which the trigger should synchronize.
`EP_TRIGGER[15:0]`	Output	Independent triggers from host.

Verilog Instantiation:

```
okTriggerIn trigIn53 (.ok1(ok1),
    .ep_addr(8'h53), .ep_clk(clk2), .ep_trigger(ep53trig));
```

VHDL Instantiation:

```
trigIn53 : okTriggerIn port map (ok1 => ok1,
    ep_addr => x"53", ep_clk => clk2, ep_trigger => ep53trig);
```

okTriggerOut

The target may trigger the host using this module. `EP_TRIGGER[15:0]` contains 16 independent trigger signals which are monitored with respect to `EP_CLK`. If `EP_TRIGGER[x]` is asserted for the rising edge of `EP_CLK`, then that trigger will be set. The next time the host checks trigger values, the triggers will be cleared.

Signal	Direction	Description
EP_ADDR[7:0]	Input	Endpoint address.
EP_CLK	Input	Clock to which the trigger is synchronized.
EP_TRIGGER[15:0]	Input	Independent triggers to host.

Verilog Instantiation:

```
okTriggerOut trigOut6A (.ok1(ok1), .ok2(ok2),
    .ep_addr(8'h6a), .ep_clk(clk2), .ep_trigger(ep6Atrig));
```

VHDL Instantiation:

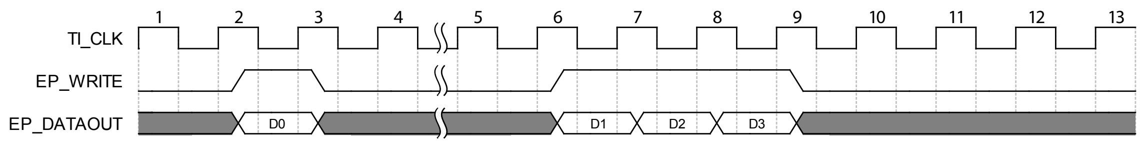
```
trigOut6A : okTriggerOut port map (ok1 => ok1, ok2 => ok2,
    ep_addr => x"6a", ep_clk => clk2, ep_trigger => ep6Atrig);
```

okPipeIn

The okPipeIn module provides a way to move synchronous multi-byte data from the host to the target. As usual, the host is the master and therefore the target must accept data as it is moved through this pipe (up to 48 MHz). The `EP_WRITE` signal is an active high signal which is asserted when data is to be accepted by the target on `EP_DATAOUT[15:0]`. It is possible that `EP_WRITE` be asserted for several consecutive cycles without deasserting. In such a case, `EP_DATAOUT[15:0]` will be changing every clock.

This somewhat simple Pipe In implementation requires that the target interface be very responsive to incoming pipe data. If the target is able to keep up with the throughput, but needs to handle data in a block fashion, coupling the okPipeIn with a FIFO (from the Xilinx CORE generator) is a good solution. Alternatively, an okBTPipeIn can be used.

The timing diagram below indicates how data is presented by the okPipeIn to user HDL. `EP_DATAOUT` contains valid data for any clock cycle where `EP_WRITE` is asserted during the rising edge of `TI_CLK`. Note that the transfer sends 4 words in this example. Although contrived, it is important to note that `EP_WRITE` may deassert during the transfer. This will generally happen with longer transfers (>256 words).



Signal	Direction	Description
`EP_ADDR[7:0]`	Input	Endpoint address.
`EP_DATAOUT[15:0]`	Output	Pipe data output.
`EP_WRITE`	Output	Active high write signal. Data should be captured when this signal is asserted.

Verilog Instantiation:

```
okPipeIn pipeIn9C (.ok1(ok1), .ok2(ok2),
    .ep_addr(8'h9c), .ep_dataout(ep9Cpipe), .ep_write(ep9Cwrite));
```

VHDL Instantiation:

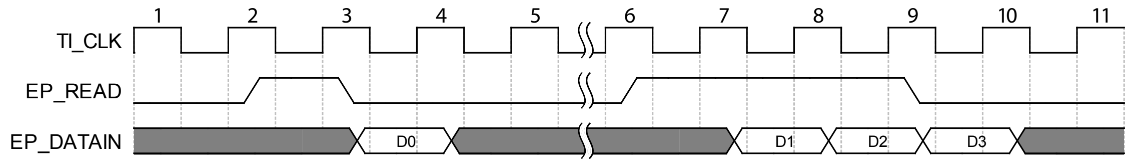
```
pipeIn9C : okPipeIn port map (ok1 => ok1, ok2 => ok2,
    ep_addr => x"9c", ep_dataout => ep9Cpipe, ep_write => ep9Cwrite);
```

okPipeOut

The okPipeOut module implements a simple version of the Pipe Out endpoint to move synchronous multi-byte data from the target to the host. Because the host is master, all reads (on the target side) occur at the host's whim. Therefore, data must be provided whenever `EP_READ` is asserted.

This simple implementation of a Pipe Out endpoint requires that the target interface be somewhat responsive to host read requests. If the target is able to keep up with the throughput, but needs to handle data in a block fashion, coupling the okPipeOut with a FIFO (from the Xilinx CORE generator) is a good solution. Alternatively, an okBTPipeOut can be used.

The timing diagram below indicates how the user HDL needs to respond to `EP_READ` with `EP_DATAIN` valid data. When `EP_READ` is asserted for the rising edge of `TI_CLK`, user HDL must respond with valid `EP_DATAIN` on the next clock edge, subject to setup and hold times appropriate for (T_{AS} and T_{AH} in the FPGA CLB timing documentation). Of course, these times are also subject to the particular routing and logic in your HDL implementation. Note that the transfer sends 4 words in this example. Although contrived, it is important to note that `EP_READ` may deassert during the transfer. This will generally happen with longer transfers (>256 words).



Signal	Direction	Description
`EP_ADDR[7:0]`	Input	Endpoint address.
`EP_DATAIN[15:0]`	Input	Pipe data input.
`EP_READ`	Output	Active-high read signal. Data must be provided in the cycle following as assertion of this signal.

Verilog Instantiation:

```
okPipeOut pipeOutA3 (.ok1(ok1), .ok2(ok2),
    .ep_addr(8'ha3), .ep_datain(epA3pipe), .ep_read(epA3read));
```

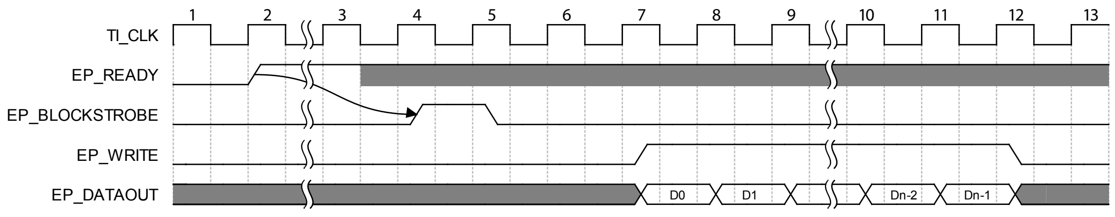
VHDL Instantiation:

```
pipeOutA3 : okPipeOut port map (ok1 => ok1, ok2 => ok2,
    ep_addr => x"a3", ep_datain => epA3pipe, ep_read => epA3read);
```

okBTPipeIn

The Block-Throttled Pipe In module is similar to the okPipeIn module, but adds two signals, EP_BLOCKSTROBE and EP_READY to handle block-level negotiation for data transfer. The host is still master, but the FPGA controls EP_READY. When EP_READY is asserted, the host is free to transmit a full block of data. When EP_READY is deasserted, the host will not transmit to the module.

EP_READY could, for example, be tied to a level indicator on a FIFO. When the FIFO has a full block of space available, it will assert EP_READY signifying that it can accept a full block transfer.



Signal	Direction	Description
EP_ADDR[7:0]	Input	Endpoint address.
EP_DATAOUT[15:0]	Output	Pipe data output.
EP_WRITE	Output	Active-high write signal. Data should be captured when this signal is asserted.
EP_BLOCKSTROBE	Output	Active-high block strobe. This is asserted for one cycle just before a block of data is written.
EP_READY	Input	Active-high ready signal. Logic should assert this signal when it is prepared to receive a full block of data.

Verilog Instantiation:

```
okBTPipeIn pipeIn9C (.ok1(ok1), .ok2(ok2)
    .ep_addr(8'h9c), .ep_dataout(ep9Cpipe), .ep_write(ep9Cwrite),
    .ep_blockstrobe(ep9Cstrobe), .ep_ready(ep9cready));
```

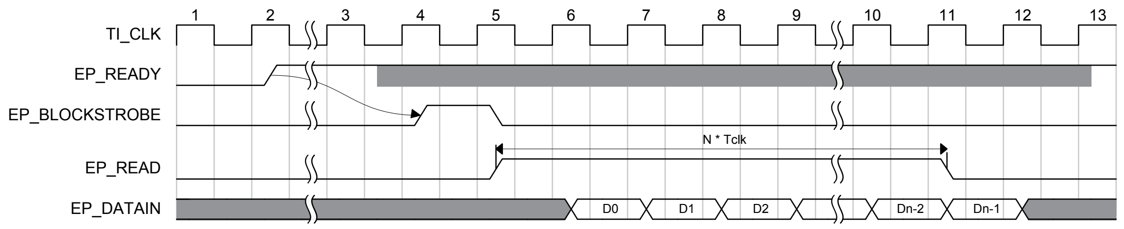
VHDL Instantiation:

```
pipeIn9C : okBTPipeIn port map (ok1 => ok1, ok2 => ok2,
    ep_addr => x"9c", ep_dataout => ep9Cpipe, ep_write => ep9Cwrite,
    ep_blockstrobe => ep9Cstrobe, ep_ready => ep9cready);
```

okBTPipeOut

The Block-Throttled Pipe Out module is similar to the okPipeOut module, but adds two signals, EP_BLOCKSTROBE and EP_READY to handle block-level negotiation for data transfer. The host is still master, but the FPGA controls EP_READY. When EP_READY is asserted, the host is free to read a full block of data. When EP_READY is deasserted, the host will not read from the module.

EP_READY could, for example, be tied to a level indicator on a FIFO. When the FIFO has a full block of data available, it will assert EP_READY signifying that a full block may be read from the FIFO.



Signal	Direction	Description
`EP_ADDR[7:0]`	Input	Endpoint address.
`EP_DATAIN[15:0]`	Input	Pipe data input.
`EP_READ`	Output	Active-high read signal. Data must be provided in the cycle following as assertion of this signal.
`EP_BLOCKSTROBE`	Output	Active-high block strobe. This is asserted for one cycle just before a block of data is read.
`EP_READY`	Input	Active-high ready signal. Logic should assert this signal when it is prepared to transmit a full block of data.

Verilog Instantiation:

```
okBTPipeOut pipeOutA3 (.ok1(ok1), .ok2(ok2)
    .ep_addr(8'ha3), .ep_datain(epA3pipe), .ep_read(epA3read),
    .ep_blockstrobe(epA3strobe), .ep_ready(epA3ready));
```

VHDL Instantiation:

```
pipeOutA3 : okBTPipeOut port map (ok1 => ok1, ok2 => ok2,
    ep_addr => x"a3", ep_datain => epA3pipe, ep_read => epA3read,
    ep_blockstrobe => epA3strobe, ep_ready => epA3ready);
```

HDL Modules - USB 3.0

FPGA Resource Requirements

The FrontPanel-enabling modules have been designed to consume as few resources as possible within the FPGA. The resource requirements for each block are listed in the tables below. Keep in mind that these are requirements for an endpoint with all bits used. In many cases, the place and route tools will optimize and remove unused components.

Resource	Slice FFs	4-in LUTs	Block RAMs
Host Interface	33	49	0
Wire In	16	14	0
Wire Out	8	5	0
Trigger In	32	21	0
Trigger Out	27	15	0
Pipe In	9	10	0
Pipe Out	0	6	0
BT Pipe In	?	?	?
BT Pipe Out	?	?	?
Register Bridge	?	?	?

Wire-OR

Multiple endpoints are attached to the `ok2` bus on the `okHost` by using a Wire-OR. Each end-point is told when it can assert its data on the bus. At all other times, it drives 0. The Wire-OR component performs a bitwise OR operation on each bit of the bus and outputs the result. In this manner, multiple endpoints can share a bus without requiring the use of tristates or a large mux.

The okWireOR is provided as a parameterized helper module in `okLibrary.v` and `okLibrary.vhd`. Please refer to the provided samples to see how to instantiate this module.

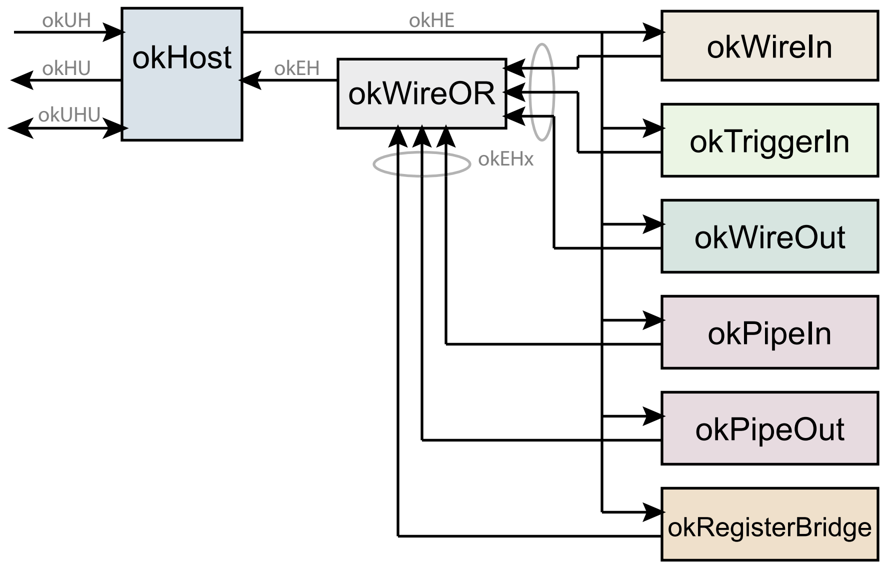
The Host Interface

The host interface is the gateway for FrontPanel to control and observe your design. It contains the logic that lets the USB microcontroller on the device communicate with the various endpoints within the design. Exactly one host interface must be instantiated in any design which uses the FrontPanel components.

The Host Interface component is the only block which is synthesized with your design. It contains a Host Interface core component (provided as a pre-synthesized module) as well as the necessary IOB components to connect to the host interface pins of the FPGA.

NOTE: The okHost is contained in `okLibrary.v` or `okLibrary.vhd`. Some details change from device to device so the exact pinouts may differ slightly from the documentation below. Please see the Opal Kelly samples in the FrontPanel installation directory for examples specific to each supported device.

The diagram below illustrates the structural relationships between the various endpoints, the okWireOR, and okHost modules.



okHost

This module must be instantiated in any design that makes use of FrontPanel virtual interface components. The following signals need to be connected directly to pins on the FPGA which go to the USB microcontroller on the device. For a listing of the pin locations for a particular product, please see the user's manual for that device.

Signal	Direction	Description
`okUH[4:0]`	Input	Host interface input signals.

Signal	Direction	Description
`okHU[2:0]`	Output	Host interface output signals.
`okUHU[31:0]`	In/Out	Host interface bidirectional signals.
`okAA`	In/Out	Host interface bidirectional signal

The remaining ports of the okHost are connected to a shared bus inside your design. These signals are collectively referred to as the target interface bus. Each endpoint must connect to these signals for proper operation.

Signal	Direction	Description
`okHE[112:0]`	Output	Control signals to the target endpoints.
`okEH[64:0]`	Input	Control signals from the target endpoints.
`okClk`	Output	Buffered copy of the host interface clock (100.8 MHz).

Instantiation of the okHost is simple in either VHDL or Verilog. Use the templates below in your toplevel HDL design. A more detailed listing can be found later in this manual as one of the examples.

Verilog Instantiation:

```
okHost okHI (.okUH(okUH), .okHU(okHU), .okUHU(okUHU), .okAA(okAA),
    .okClk(okClk), .okHE(okHE), .okEH(okEH));
```

VHDL Instantiation:

```
okHI : okHost port map (okUH => okUH, okHU => okHU, okUHU =>okUHU, okAA => okAA,
    okClk => okClk, okHE => okHE, okEH => okEH);
```

Each endpoint is connected to 48 target interface pins on the okHost module. The direction is from the perspective of the endpoint module.

Signal	Direction	Description
`okHE[112:0]`	Input	Interface control (host to endpoint)
`okEH[64:0]`	Output	Interface control (endpoint to host)

These signals are present in every endpoint. In the signal tables for the independent endpoints below, we have left out these common signals.

okWireIn

In addition to the target interface pins, the okWireIn adds a single 32-bit output bus called `EP_DATAOUT[31:0]`. The pins of this bus are connected to your design as wires and act as asynchronous connections from FrontPanel components to your HDL.

When FrontPanel updates the Wire Ins, it writes new values to the wires, then updates them all at the same time. Therefore, although the wires are asynchronous endpoints, they are all updated at the same time *on the host interface clock*.

Signal	Direction	Description
`EP_DATAOUT[31:0]`	Output	Wire values output. (sent from host)

Verilog Instantiation:

```
okWireIn wire03 (.okHE(okHE),
    .ep_addr(8'h03), .ep_dataout(ep03data));
```

VHDL Instantiation:

```
wire03 : okWireIn port map (okHE => okHE,
    ep_addr => x"03", ep_dataout => ep03data);
```

okWireOut

An okWireOut module adds a single 16-bit input bus called `EP_DATAIN`. Signals on these pins are read whenever FrontPanel updates the state of its wire values. In fact, all wires are captured simultaneously (synchronous to the host interface clock) and read out sequentially.

Signal	Direction	Description
`EP_DATAIN[31:0]`	Input	Wire values input. (to be sent to host)

Verilog Instantiation:

```
okWireOut wire21 (.okHE(okHE), .okEH(okEH),
    .ep_addr(8'h21), .ep_datain(ep21data));
```

VHDL Instantiation:

```
wire21 : okWireOut port map (okHE => okHE, okEH => okEH,
    ep_addr => x"21", ep_datain => ep21data);
```

okTriggerIn

The okTriggerIn provides `EP_CLK` and `EP_TRIGGER` as interface signals. The Trigger In endpoint produces a single-cycle trigger pulse on any of `EP_TRIGGER` which is synchronized to the clock signal `EP_CLK`. Therefore, the single-cycle does not necessarily have to be a single host interface cycle. Rather, the module takes care of crossing the clock boundary properly.

Signal	Direction	Description
`EP_ADDR[7:0]`	Input	Endpoint address.
`EP_CLK`	Input	Clock to which the trigger should synchronize.
`EP_TRIGGER[31:0]`	Output	Independent triggers from host.

Verilog Instantiation:

```
okTriggerIn trigIn53 (.okHE(okHE),
    .ep_addr(8'h53), .ep_clk(clk2), .ep_trigger(ep53trig));
```

VHDL Instantiation:

```
trigIn53 : okTriggerIn port map (okHE => okHE,
    ep_addr => x"53", ep_clk => clk2, ep_trigger => ep53trig);
```

okTriggerOut

The target may trigger the host using this module. `EP_TRIGGER[15:0]` contains 16 independent trigger signals which are monitored with respect to `EP_CLK`. If `EP_TRIGGER[x]` is asserted for the rising edge of EP_CLK, then that trigger will be set. The next time the host checks trigger values, the triggers will be cleared.

Signal	Direction	Description
EP_ADDR[7:0]	Input	Endpoint address.
EP_CLK	Input	Clock to which the trigger is synchronized.
EP_TRIGGER[31:0]	Input	Independent triggers to host.

Verilog Instantiation:

```
okTriggerOut trigOut6A (.okHE(okHE), .okEH(okEH),
    .ep_addr(8'h6a), .ep_clk(clk2), .ep_trigger(ep6Atrig));
```

VHDL Instantiation:

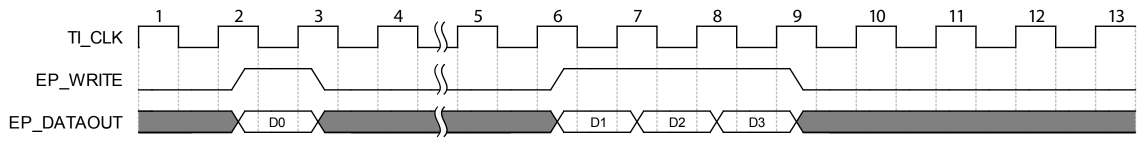
```
trigOut6A : okTriggerOut port map (okHE => okHE, okEH => okEH,
    ep_addr => x"6a", ep_clk => clk2, ep_trigger => ep6Atrig);
```

okPipeIn

The okPipeIn module provides a way to move synchronous multi-byte data from the host to the target. As usual, the host is the master and therefore the target must accept data as it is moved through this pipe (up to 100.8 MHz). The EP_WRITE signal is an active high signal which is asserted when data is to be accepted by the target on EP_DATAOUT[31:0]. It is possible that EP_WRITE be asserted for several consecutive cycles without deasserting. In such a case, EP_DATAOUT[31:0] will be changing every clock.

This somewhat simple Pipe In implementation requires that the target interface be very responsive to incoming pipe data. If the target is able to keep up with the throughput, but needs to handle data in a block fashion, coupling the okPipeIn with a FIFO (from the Xilinx CORE generator) is a good solution. Alternatively, an okBTPipeIn can be used.

The timing diagram below indicates how data is presented by the okPipeIn to user HDL. EP_DATAOUT contains valid data for any clock cycle where EP_WRITE is asserted during the rising edge of TI_CLK. Note that the transfer sends 4 words in this example. Although contrived, it is important to note that EP_WRITE may deassert during the transfer. This will generally happen with longer transfers (>256 words).



Signal	Direction	Description
EP_ADDR[7:0]	Input	Endpoint address.
EP_DATAOUT[31:0]	Output	Pipe data output.
EP_WRITE	Output	Active high write signal. Data should be captured when this signal is asserted.

Verilog Instantiation:

```
okPipeIn pipeIn9C (.okHE(okHE), .okEH(okEH),
    .ep_addr(8'h9c), .ep_dataout(ep9Cpipe), .ep_write(ep9Cwrite));
```

VHDL Instantiation:

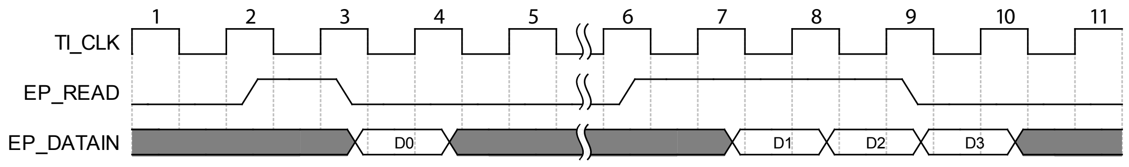
```
pipeIn9C : okPipeIn port map (okHE => okHE, okEH => okEH,
    ep_addr => x"9c", ep_dataout => ep9Cpipe, ep_write => ep9Cwrite);
```

okPipeOut

The okPipeOut module implements a simple version of the Pipe Out endpoint to move synchronous multi-byte data from the target to the host. Because the host is master, all reads (on the target side) occur at the host's whim. Therefore, data must be provided whenever EP_READ is asserted.

This simple implementation of a Pipe Out endpoint requires that the target interface be somewhat responsive to host read requests. If the target is able to keep up with the throughput, but needs to handle data in a block fashion, coupling the okPipeOut with a FIFO (from the Xilinx CORE generator) is a good solution. Alternatively, an okBTPipeOut can be used.

The timing diagram below indicates how the user HDL needs to respond to EP_READ with EP_DATAIN valid data. When EP_READ is asserted for the rising edge of TI_CLK, user HDL must respond with valid EP_DATAIN on the next clock edge, subject to setup and hold times appropriate for (T_{AS} and T_{AH} in the FPGA CLB timing documentation). Of course, these times are also subject to the particular routing and logic in your HDL implementation. Note that the transfer sends 4 words in this example. Although contrived, it is important to note that EP_READ may deassert during the transfer. This will generally happen with longer transfers (>256 words).



Signal	Direction	Description
EP_ADDR[7:0]	Input	Endpoint address.
EP_DATAIN[31:0]	Input	Pipe data input.
EP_READ	Output	Active-high read signal. Data must be provided in the cycle following as assertion of this signal.

Verilog Instantiation:

```
okPipeOut pipeOutA3 (.okHE(okHE), .okEH(okEH),
    .ep_addr(8'ha3), .ep_datain(epA3pipe), .ep_read(epA3read));
```

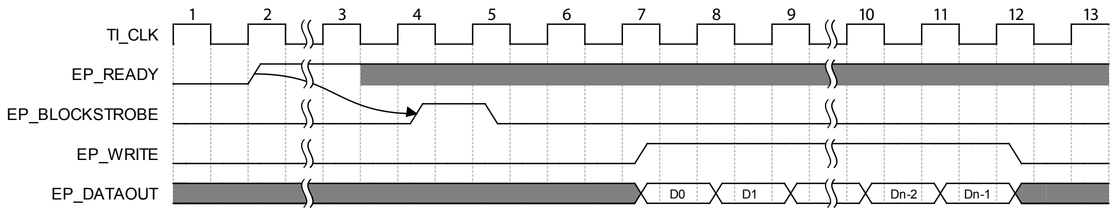
VHDL Instantiation:

```
pipeOutA3 : okPipeOut port map (okHE => okHE, okEH => okEH,
    ep_addr => x"a3", ep_datain => epA3pipe, ep_read => epA3read);
```

okBTPipeIn

The Block-Throttled Pipe In module is similar to the okPipeIn module, but adds two signals, EP_BLOCKSTROBE and EP_READY to handle block-level negotiation for data transfer. The host is still master, but the FPGA controls EP_READY. When EP_READY is asserted, the host is free to transmit a full block of data. When EP_READY is deasserted, the host will not transmit to the module.

EP_READY could, for example, be tied to a level indicator on a FIFO. When the FIFO has a full block of space available, it will assert EP_READY signifying that it can accept a full block transfer.



Signal	Direction	Description
EP_ADDR[7:0]	Input	Endpoint address.
EP_DATAOUT[31:0]	Output	Pipe data output.
EP_WRITE	Output	Active-high write signal. Data should be captured when this signal is asserted.
EP_BLOCKSTROBE	Output	Active-high block strobe. This is asserted for one cycle just before a block of data is written.
EP_READY	Input	Active-high ready signal. Logic should assert this signal when it is prepared to receive a full block of data.

Verilog Instantiation:

```
okBTPipeIn pipeIn9C (.okHE(okHE), .okEH(okEH),
    .ep_addr(8'h9c), .ep_dataout(ep9Cpipe), .ep_write(ep9Cwrite),
    .ep_blockstrobe(ep9Cstrobe), .ep_ready(ep9cready));
```

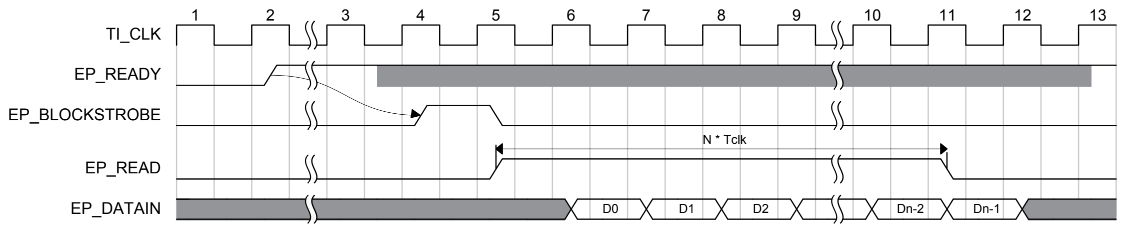
VHDL Instantiation:

```
pipeIn9C : okBTPipeIn port map (okHE => okHE, okEH => okEH,
    ep_addr => x"9c", ep_dataout => ep9Cpipe, ep_write => ep9Cwrite,
    ep_blockstrobe => ep9Cstrobe, ep_ready => ep9cready);
```

okBTPipeOut

The Block-Throttled Pipe Out module is similar to the okPipeOut module, but adds two signals, EP_BLOCKSTROBE and EP_READY to handle block-level negotiation for data transfer. The host is still master, but the FPGA controls EP_READY. When EP_READY is asserted, the host is free to read a full block of data. When EP_READY is deasserted, the host will not read from the module.

EP_READY could, for example, be tied to a level indicator on a FIFO. When the FIFO has a full block of data available, it will assert EP_READY signifying that a full block may be read from the FIFO.



Signal	Direction	Description
`EP_ADDR[7:0]`	Input	Endpoint address.
`EP_DATAIN[31:0]`	Input	Pipe data input.
`EP_READ`	Output	Active-high read signal. Data must be provided in the cycle following as assertion of this signal.
`EP_BLOCKSTROBE`	Output	Active-high block strobe. This is asserted for one cycle just before a block of data is read.
`EP_READY`	Input	Active-high ready signal. Logic should assert this signal when it is prepared to transmit a full block of data.

Verilog Instantiation:

```
okBTPipeOut pipeOutA3 (.okHE(okHE), .okEH(okEH),
    .ep_addr(8'ha3), .ep_datain(epA3pipe), .ep_read(epA3read),
    .ep_blockstrobe(epA3strobe), .ep_ready(epA3ready));
```

VHDL Instantiation:

```
pipeOutA3 : okBTPipeOut port map (okHE => okHE, okEH => okEH,
    ep_addr => x"a3", ep_datain => epA3pipe, ep_read => epA3read,
    ep_blockstrobe => epA3strobe, ep_ready => epA3ready);
```

okRegisterBridge

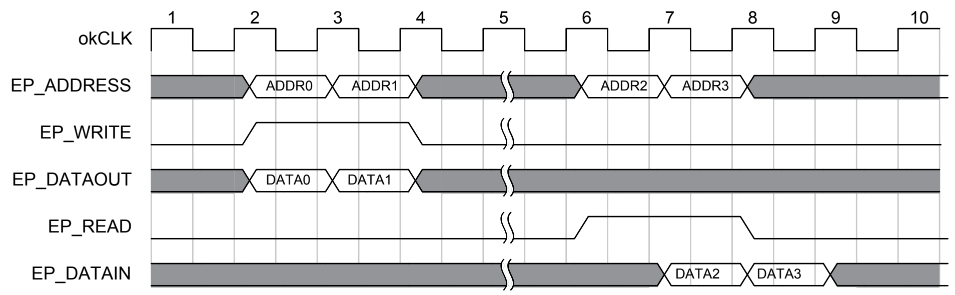
The Register Bridge provides synchronous register file access to a 32-bit data, 32-bit address register space. User HDL is responsible for responding to register reads and writes, but is free to interpret the address in any manner.

Register Writes

A register write is signalled with the one-cycle assertion of `EP_WRITE`. When `EP_WRITE` is asserted, user HDL should capture both the address (on `EP_ADDRESS`) and data (on `EP_DATAOUT`). Register writes may occur to any address in any clock cycle, including multiple consecutive cycles.

Register Reads

A register read is signalled with the one-cycle assertion of `EP_READ`. When `EP_READ` is asserted, user HDL should capture the address (`EP_ADDRESS`) and respond in the next clock cycle by driving `EP_DATAIN` with the value requested. Register reads may be requested for any address in any clock cycle, including multiple consecutive cycles.



Signal	Direction	Description
`EP_WRITE`	Output	Asserted during a write cycle.
`EP_READ`	Output	Asserted during a read cycle.
`EP_ADDRESS[31:0]`	Output	Driven with the requested address during read and write cycles.
`EP_DATAOUT[31:0]`	Output	Driven with valid data during a write cycle.
`EP_DATAIN[31:0]`	Input	This value is captured in the cycle following a read cycle.

Verilog Instantiation:

```
okRegisterBridge regBridge (.okHE(okHE), .okEH(okEH),
    .ep_write(regWrite), .ep_read(regRead), .ep_address(regAddress),
    .ep_dataout(regDataOut), .ep_datain(regDataIn));
```

VHDL Instantiation:

```
regBridge : okRegisterBridge port map (okHE => okHE, okEH => okEH,
    ep_write => regWrite, ep_read => regRead, ep_address => regAddress,
    ep_dataout => regDataOut, ep_datain => regDataIn);
```

HDL Modules - PCI Express

FPGA Resource Requirements

The FrontPanel-enabling modules have been designed to consume as few resources as possible within the FPGA. The resource requirements for each block are listed in the tables below. Keep in mind that these are requirements for an endpoint with all bits used. In many cases, the place and route tools will optimize and remove unused components.

Resource	Slice FFs	LUTs	Block RAMs
Host	2365	2755	8
Wire In	64	4	0
Wire Out	32	35	0
Trigger In	128	68	0
Trigger Out	99	69	0
Pipe In	41	44	2
Pipe Out	41	44	2

The resource requirements above do not include the resources required by instantiated FIFOs except for the Block RAM requirements.

Wire-OR

Multiple endpoints are attached to the `okEH` bus on the `okHost` by using a Wire-OR. Each endpoint is told when it can assert its data on the bus. At all other times, it drives 0. The Wire-OR component performs a bitwise OR operation on each bit of the bus and outputs the result. In this manner, multiple endpoints can share a bus without requiring the use of tristates or a large mux.

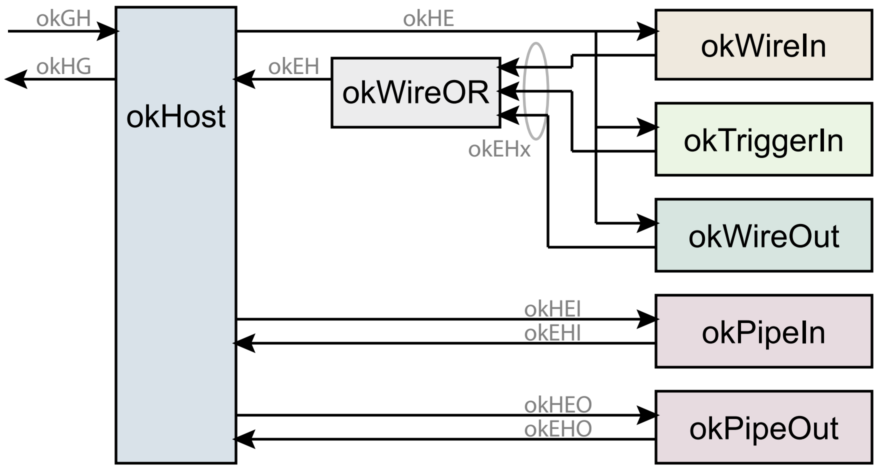
The okWireOR is provided as a parameterized helper module in `okLibrary.v` and `okLibrary.vhd`. Please refer to the provided samples to see how to instantiate this module.

The Host Interface

The host interface is the gateway for FrontPanel to control and observe your design. It contains the logic that communicates with the PCI Express bridge. Exactly one host interface must be instantiated in any design which uses the FrontPanel components.

The okHost component is the only block which is synthesized with your design. It contains an okCoreHarness component (provided as a pre-synthesized module) as well as the necessary IOB components to connect to the host interface pins of the FPGA.

The diagram below illustrates the structural relationships between the various endpoints, the okWireOR, and okHost modules.



okHost

This module must be instantiated in any design that makes use of FrontPanel virtual interface components. The following signals need to be connected directly to pins on the FPGA which go to the USB microcontroller on the XEM. For a listing of the pin locations for a particular XEM product, please see the user's manual for that device.

Signal	Direction	Description
`okGH[28:0]`	Input	Input to the host interface from the PCI Express bridge
`okHG[27:0]`	Output	Output from the host interface to the PCI Express bridge

The remaining ports of the okHost are connected to endpoints inside your design. These signals are collectively referred to as the target interface bus. Endpoints connect to one or more of these signals for proper operation.

Signal	Direction	Description
`okEH[32:0]`	Input	Endpoint - to - Host signals (Wires, Triggers)
`okEHI[37:0]`	Input	Endpoint - to - Host signals (for Pipe In)
`okEHO[102:0]`	Input	Endpoint - to - Host signals (for Pipe Out)

Signal	Direction	Description
`okHE[46:0]`	Output	Host - to - Endpoint signals (Wires, Triggers)
`okHEI[99:0]`	Output	Host - to - Endpoint signals (for Pipe In)
`okHEO[43:0]`	Output	Host - to - Endpoint signals (for Pipe Out)
`ti_clk`	Output	Copy of the host interface 50 MHz clock

Instantiation of the okHost is simple in either VHDL or Verilog. Use the templates below in your toplevel HDL design. A more detailed listing can be found later in this manual as one of the examples. If pipes are not used in your design, you can force the inputs (`okEHI` and `okEHO`) to all 0's and leave the outputs (`okHEI` and `okHEO`) unconnected.

Verilog Instantiation:

```
okHost_XEM6110 hostIF (.okGH(okGH), .okHG(okHG), ..., .ti_clk(ti_clk));
```

VHDL Instantiation:

```
okHI : okHost_XEM6110 port map (okGH => okGH, okHG => okHG, ...
    ti_clk => ticlk);
```

okWireIn

In addition to the target interface pins, the okWireIn adds a single 32-bit output bus called `EP_DATAOUT[31:0]`. The pins of this bus are connected to your design as wires and act as asynchronous connections from FrontPanel components to your HDL.

When FrontPanel updates the Wire Ins, it writes new values to the wires, then updates them all at the same time. Therefore, although the wires are asynchronous endpoints, they are all updated at the same time *on the host interface clock*.

Signal	Direction	Description
`EP_DATAOUT[31:0]`	Output	Wire values output. (sent from host)

Verilog Instantiation:

```
okWireIn wire03 (.ok1(okHE), .ep_addr(8'h03), .ep_dataout(ep03data));
```

VHDL Instantiation:

```
wire03 : okWireIn port map (ok1 => okHE,
    ep_addr => x"03", ep_dataout => ep03data);
```

okWireOut

An okWireOut module adds a single 32-bit input bus called `EP_DATAIN[31:0]`. Signals on these pins are read whenever FrontPanel updates the state of its wire values. In fact, all wires are captured simultaneously (synchronous to the host interface clock) and read out sequentially.

Signal	Direction	Description
`EP_DATAIN[31:0]`	Input	Wire values input. (to be sent to host)

Verilog Instantiation:

```
okWireOut wire21 (.ok1(okHE), .ok2(okEHx),
    .ep_addr(8'h21), .ep_datain(ep21data));
```

VHDL Instantiation:

```
wire21 : okWireOut port map (ok1 => ok1, ok2 => ok2,
    ep_addr => x"21", ep_datain => ep21data);
```

okTriggerIn

The okTriggerIn provides `EP_CLK` and `EP_TRIGGER[31:0]` as interface signals. The Trigger In endpoint produces a single-cycle trigger pulse on any of `EP_TRIGGER[31:0]` which is synchronized to the clock signal `EP_CLK`. Therefore, the single-cycle does not necessarily have to be a single host interface cycle. Rather, the module takes care of crossing the clock boundary properly.

Signal	Direction	Description
`EP_ADDR[7:0]`	Input	Endpoint address.
`EP_CLK`	Input	Clock to which the trigger should synchronize.
`EP_TRIGGER[31:0]`	Output	Independent triggers from host.

Verilog Instantiation:

```
okTriggerIn trigIn53 (.ok1(okHE),
    .ep_addr(8'h53), .ep_clk(clk2), .ep_trigger(ep53trig));
```

VHDL Instantiation:

```
trigIn53 : okTriggerIn port map (ok1 => okHE,
    ep_addr => x"53", ep_clk => clk2, ep_trigger => ep53trig);
```

okTriggerOut

The target may trigger the host using this module. `EP_TRIGGER[31:0]` contains 32 independent trigger signals which are monitored with respect to `EP_CLK`. If `EP_TRIGGER[x]` is asserted for the rising edge of `EP_CLK`, then that trigger will be set. The next time the host checks trigger values, the triggers will be cleared.

Signal	Direction	Description
`EP_ADDR[7:0]`	Input	Endpoint address.
`EP_CLK`	Input	Clock to which the trigger is synchronized.
`EP_TRIGGER[31:0]`	Input	Independent triggers to host.

Verilog Instantiation:

```
okTriggerOut trigOut6A (.ok1(okHE), .ok2(okEHx),
    .ep_addr(8'h6a), .ep_clk(clk2), .ep_trigger(ep6Atrig));
```

VHDL Instantiation:

```
trigOut6A : okTriggerOut port map (ok1 => okHE, ok2 => okEHx,
    ep_addr => x"6a", ep_clk => clk2, ep_trigger => ep6Atrig);
```

okPipeIn

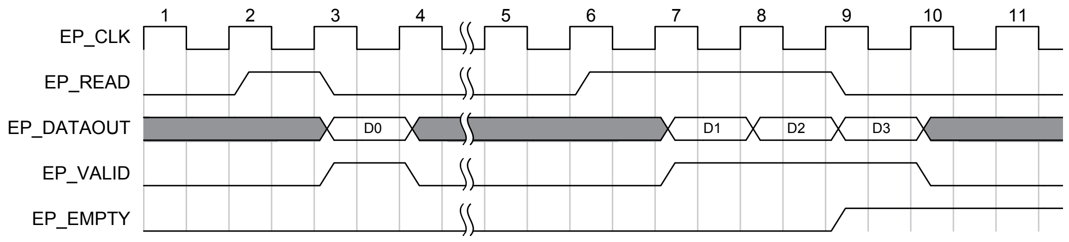
The okPipeIn module provides a way to move synchronous multi-byte data from the host to the target. As usual, the host is the master and initiates all transfers. Therefore the target should be ready to accept data as it is moved through this pipe (up to 50 MHz). A small FIFO (511 words) is built into the okPipeIn to allow some flexibility, but it is generally assumed that the transfer will run to completion expeditiously.

`EP_CLK` may be independent of the host interface clock. When the API initiates a pipe transfer (`WriteToPipeIn`), `EP_START` wil be asserted for a single `EP_CLK` cycle. This is to indicate to user HDL that the transfer has started. User HDL may reset the FIFO (empty it) at any time but requires at least three clock cycles to complete.. `EP_FIFO_RESET` may be tied to `EP_START` for convenient operation.

The `EP_EMPTY` signal will deassert when data is available in the FIFO. User code should then assert `EP_READ` to read each available word until `EP_EMPTY` is deasserted. `EP_VALID` is used to indicate when valid data is available on `EP_DATA`. `EP_COUNT[8:0]` pessimistically indicates the number of words remianing in the FIFO. It may temporarily under-report the count, but will never over-report so that underflow is avoided.

At the end of the complete transfer, `EP_DONE` is asserted for one `EP_CLK` cycle.

The timing diagram below indicates how data is presented by the okPipeIn to user HDL. In this case, 4 data words have been placed into the FIFO by the okHost. These four data words are read out with a brief pause inserted for illustration. `EP_EMPTY` is shown coincident with the last word read out and remains asserted to indicate that no more data is available. Any reads at this point will cause an underflow condition (not indicated) and `EP_VALID` will not be asserted.



Signal	Direction	Description
`okHEI[99:0]`	Input	Host - to - endpoint control signals.
`okEHI[37:0]`	Output	Endpoint - to - host control signals.
`EP_CLK`	Input	Endpoint-side clock to FIFO.
`EP_START`	Output	Indicates the start of a transfer.
`EP_DONE`	Output	Indicates the completion of a transfer.
`EP_FIFO_RESET`	Input	FIFO reset signal.
`EP_READ`	Input	Application asserts this to read a word.
`EP_DATA[63:0]`	Output	Pipe data output.
`EP_VALID`	Output	Asserted one cycle after a successful read.
`EP_COUNT[8:0]`	Output	Indicates the number of words available in the FIFO.
`EP_EMPTY`	Output	Asserted after the last successful read. Remains asserted while the FIFO is empty.

Verilog Instantiation:

```
okPipeIn pipeIn (.okHEI(okHEI), .okEHI(okEHI),
    .ep_clk(pipeInClk), .ep_start(pipeInStart), .epdone(pipeInDone),
    .ep_fifo_reset(pipeInReset), .ep_read(pipeInRead), .ep_data(pipeInData),
    .ep_valid(pipeInValid), .ep_count(pipeInCount), .ep_empty(pipeInEmpty));
```

VHDL Instantiation:

```
pipeIn : okPipeIn port map (okHEI => okHEI, okEHI => okEHI,
    ep_clk => pipeInClk, ep_start => pipeInStart, ep_done => pipeInDone,
    ep_fifo_reset => pipeInReset, ep_read => pipeInRead, ep_data => pipeInData,
    ep_valid => pipeInValid, ep_count => pipeInCount, ep_empty => pipeInEmpty);
```

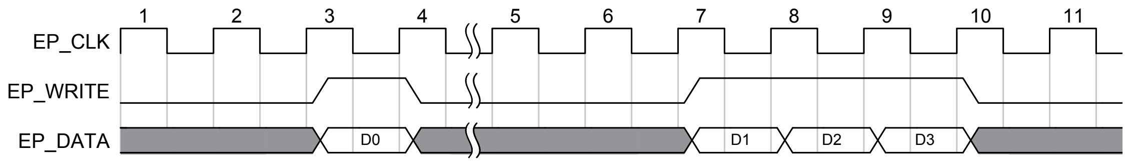
okPipeOut

The okPipeOut module provides a way to move synchronous multi-byte data from the target to the host. As usual, the host is the master and initiates all transfers. The target should be ready to provide data when it is requested. A small FIFO (511 words) is built into the okPipeOut to allow some flexibility, but the transfer will run to completion in all cases. If user HDL is unable to keep up with the transfer, the FIFO will underrun. This error condition will be reported to user software, but the transfer will complete and the data following underrun will be unreliable.

`EP_CLK` may be independent of the host interface clock. When the API initiates a pipe transfer (`ReadFromPipeOut`), `EP_START` wil be asserted for a single `EP_CLK` cycle. This is to indicate to user HDL that the transfer has started. User HDL may start filling the transfer FIFO at this time. User HDL may reset the FIFO at any time, but requires at least three clock cycles to complete. During this time, `EP_FULL` will be asserted.. `EP_FIFO_RESET` may be tied to `EP_START` for convenient operation.

`EP_FULL` will assert when the FIFO is unable to accept data. `EP_COUNT` indicates the number of words currently in the FIFO and may also be used by the user HDL to determine when to provide more data.

The timing diagram below illustrates the communication.



Signal	Direction	Description
`okHEO[43:0]`	Input	Host - to - endpoint control signals.
`okEHO[102:0]`	Output	Endpoint - to - host control signals.
`EP_CLK`	Input	Endpoint-side clock to the FIFO.
`EP_START`	Output	Indicates the start of a transfer.
`EP_DONE`	Output	Indicates the completion of a transfer.
`EP_FIFO_RESET`	Input	FIFO reset signal.
`EP_WRITE`	Input	FIFO write signal to enter a single data word.
`EP_DATA[63:0]`	Input	Pipe data input.
`EP_COUNT[8:0]`	Output	Indicates the number of words in the FIFO.

Signal	Direction	Description
`EP_FULL`	Output	Asserted when the FIFO is full.

Verilog Instantiation:

```
okPipeOut pipeOut (.okHEO(okHEO), .okEHO(okEHO),
    .ep_clk(pipeOutClk), .ep_start(pipeOutStart), .ep_done(pipeOutDone),
    .ep_fifo_reset(pipeOutReset), .ep_write(pipeOutWrite), .ep_data(pipeOutData),
    .ep_count(pipeOutCount), .ep_full(pipeOutFull));
```

VHDL Instantiation:

```
pipeOut : okPipeOut port map (okHEO => okHEO, okEHO => okEHO,
    ep_clk => pipeOutClk, ep_start => pipeOutStart, ep_done => pipeOutDone,
    ep_fifo_reset=>pipeOutReset, ep_write=>pipeOutWrite, ep_data => pipeOutData,
    ep_count => pipeOutCount, ep_full => pipeOutFull);
```

Using the FrontPanel Application

FrontPanel provides essential functionality to make using FrontPanel devices easy and intuitive to use. This functionality includes downloading FPGA configuration files and configuring the on-board peripherals for use in a design, but it also extends to loading "FrontPanel Profiles" to control and interface to your design.

Main Window

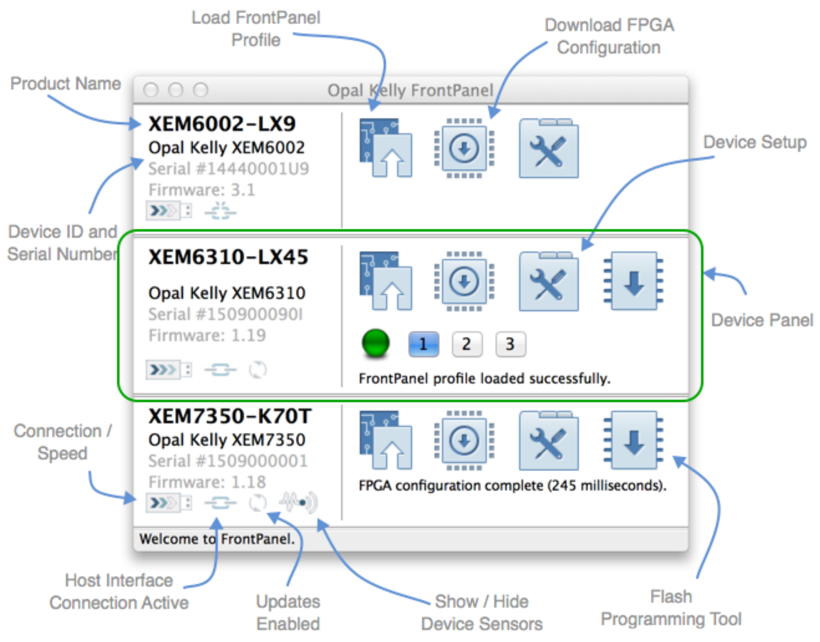
The FrontPanel interface has a simple presentation as shown on the next page with three attached devices.

Available devices are shown in individual "Device Panels" with each panel providing functionality specifically for that device. Device Panels are automatically updated as USB devices are added and removed from the bus. If you have BIOS-supported PCI Express hot-plug (e.g. ExpressCard on laptops), then PCI Express updates also occur.

The left side of the Device Panel is populated with information such as the product name, the user-specified Device ID, serial number, and firmware. The Device ID is a clickable label. When clicked, a dialog will appear allowing you to change the Device ID.

Below the device information is a small icon which will be colored (blue and green) when the FPGA on the device is enabled with the FrontPanel Host Interface. When a Host Interface is not detected, the icon will display in gray.

Icons to the right of the device information are described in the following sections.



Opal Kelly FrontPanel

XEM6002-LX9
Opal Kelly XEM6002
Serial #14440001U9
Firmware: 3.1

XEM6310-LX45
Opal Kelly XEM6310
Serial #150900090I
Firmware: 1.19
1 2 3
FrontPanel profile loaded successfully.

XEM7350-K70T
Opal Kelly XEM7350
Serial #1509000001
Firmware: 1.18
FPGA configuration complete (245 milliseconds).

Welcome to FrontPanel.

Load a FrontPanel Profile

A FrontPanel "Profile" is an XML file with the extension .XFP. The profile describes one or more Interface Panels which communicate with your device. A new profile may be loaded at any time, but only one profile for each device is available at any time. That is, the previous profile is unloaded before loading in the new one. You can load a new profile by clicking on the button shown at the left. A file selector dialog will open asking you to select a profile.

When a selection is confirmed, the profile is loaded and the first panel is displayed. If there are more panels in the profile, they will not be displayed. However, a toggle button is displayed in the "Panel Selections" area for each Interface Panel to activate (or deactivate) a specific panel. To open another panel, simply click that panel's button on the list.

Panel Identification

The colored sphere next to the Panel Selection buttons has a drop-down menu to show or hide all panels, unload the FrontPanel Profile, and change the associated color. The color of the sphere is matched to the color of spheres in the status area for any panels open from the active FrontPanel Profile. When multiple devices are attached, the spheres help quickly identify Interface Panels with the appropriate device.

Drag and Drop

As an alternative to opening the file dialog to load a new profile, you can drag an XFP file and drop it on the button. This will load the profile and open the first panel just like opening the file through the file selector.

FPGA Configuration Download

To download an FPGA configuration file to the target device, simply click on the icon shown to the left. A file selector dialog will appear from which you can choose the Xilinx bitfile to download. If you accept the file, the download will proceed immediately. Four things happen when you configure the device:

1. The on-board PLL is configured with the parameters stored in EEPROM.
2. The FPGA is reset and a programming sequence is initiated.
3. The configuration data is downloaded to the FPGA.
4. The FPGA is checked to verify that the configuration was successful (DONE is asserted).

Once complete, the FPGA is now configured and "running" with the new design.

Drag and Drop

As an alternative to clicking the download icon and using a file selector to choose the configuration file, you can simply drag a Xilinx bitfile onto the icon and release it. FrontPanel then proceeds as if you had just chosen the file in the file selector.

Device Setup

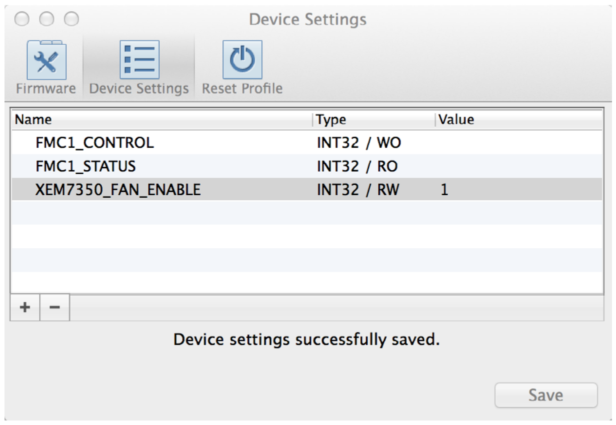
Several device configuration interfaces are available from the Device Setup icon. Available configuration interfaces depend on the specific device attached and the features it supports.

Firmware Information

Various information is available about the device firmware version on this page. If the device is an FMC carrier such as the Shuttle LX1 (XEM6006) or Shuttle TX1 (XEM7350), peripheral information is also available if the peripheral has an IPMI EEPROM.

Device Settings

USB 3.0 devices support Device Settings that may be either non-volatile (persistent settings stored in Flash memory) or volatile (settings that monitor or control device behavior while power is available). These settings may be viewed and edited on this page.



Name	Type	Value
FMC1_CONTROL	INT32 / WO	
FMC1_STATUS	INT32 / RO	
XEM7350_FAN_ENABLE	INT32 / RW	1

Device settings successfully saved.

Reset Profile

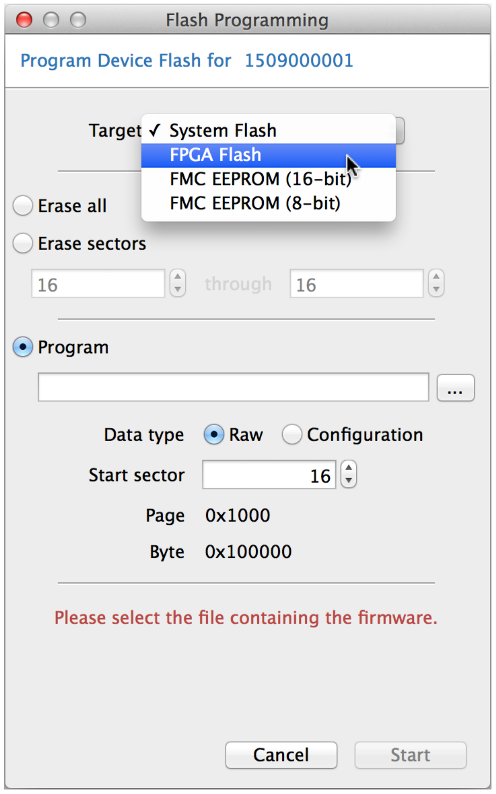
USB 3.0 devices support Reset Profiles (see the Reset Profiles section in this document). Configuring these reset profiles may be done through this user interface.

Flash Programming Tool

FrontPanel is able to program the on-board Flash memory for supported devices. The typical application for Flash programming is downloading an FPGA configuration bitfile to the Flash to allow the device to boot the FPGA on power-up in a "non-tethered" application. The host interface may be used after boot, if required.

On-board Flash memory is classified as one of two types:

- System Flash is available to the host controller and may be accessed without an active FPGA configuration. This memory may not be accessed directly by the FPGA. System Flash is only availble on USB 3.0 devices.
- FPGA Flash is connected to the FPGA. It is accessible only to the host through the use of an FPGA configuration that supports communication between the host and the memory.



Program Device Flash for 1509000001

Erase sectors: 16 through 16

[✓ Program]

Data type [✓ Raw] Configuration

Start sector 16

Page 0x1000

Byte 0x100000

Please select the file containing the firmware.

System Flash Programming

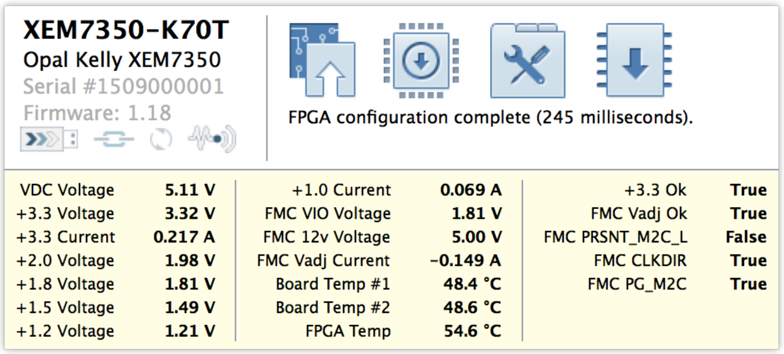
FrontPanel uses the `FlashErase`, `FlashWrite`, and `FlashRead` APIs to erase and program the System Flash memory, if available. Please see the corresponding Device User's Manual for available size and memory layout information which is also available via the `DeviceInfo` API structure.

FPGA Flash Programming

Programming the FPGA Flash requires FrontPanel to configure the FPGA with a special bitfile that allows FrontPanel to access the Flash memory. This bitfile is first downloaded to the FPGA before the Flash erase or programming steps are performed.

Device Sensors Panel

When connected to supported devices, the Device Sensors icon will appear. Click on this icon to toggle the Device Sensors panel. The panel displays all available device sensors and their corresponding values. The panel is automatically updated periodically.



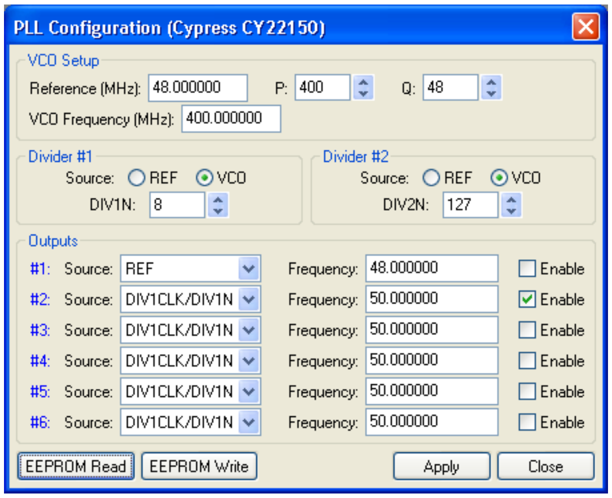
XEM7350-K70T
Opal Kelly XEM7350
Serial #1509000001
Firmware: 1.18

FPGA configuration complete (245 milliseconds).

VDC Voltage	5.11 V	+1.0 Current	0.069 A	+3.3 Ok	True
+3.3 Voltage	3.32 V	FMC VIO Voltage	1.81 V	FMC Vadj Ok	True
+3.3 Current	0.217 A	FMC 12v Voltage	5.00 V	FMC PRSNT_M2C_L	False
+2.0 Voltage	1.98 V	FMC Vadj Current	-0.149 A	FMC CLKDIR	True
+1.8 Voltage	1.81 V	Board Temp #1	48.4 °C	FMC PG_M2C	True
+1.5 Voltage	1.49 V	Board Temp #2	48.6 °C		
+1.2 Voltage	1.21 V	FPGA Temp	54.6 °C		

PLL Configuration (CY22150)

The on-board PLL is available to the USB microcontroller as an I²C peripheral. Through FrontPanel, you can configure the PLL using the PLL Configuration Dialog which is opened by clicking on the icon to the left. When you do so, the current PLL configuration is read and the following dialog appears:



VCO Setup

Reference (MHz): 48.000000 P: 400 Q: 48

VCO Frequency (MHz): 400.000000

Divider #1 — Source: REF [✓ VCO] DIV1N: 8

Divider #2 — Source: REF [✓ VCO] DIV2N: 127

Outputs

	Source	Frequency	Enable
#1:	REF	48.000000	☐
#2:	DIV1CLK/DIV1N	50.000000	✓
#3:	DIV1CLK/DIV1N	50.000000	☐
#4:	DIV1CLK/DIV1N	50.000000	☐
#5:	DIV1CLK/DIV1N	50.000000	☐
#6:	DIV1CLK/DIV1N	50.000000	☐

As you make changes in the PLL Configuration Dialog, the output frequencies are automatically updated to indicate how the outputs will behave with the current selections.

Details of the PLL configuration are available in Cypress documentation for the CY22150. A brief description of the parameters follows.

VCO Setup

The CY22150 contains a single PLL which is used as the source to a divider network which then produces the signals at the output. Because of this, all outputs are referenced from the same PLL. The VCO frequency is produced by dividing the reference frequency (fixed at 48 MHz for the XEM3001) by Q and multiplying by P. Cypress specifies that the VCO frequency should be kept between 250 kHz and 400 MHz for reliable operation.

The valid range for P is 8 to 2055. The valid range for Q is 2 to 129.

Divider #1 and #2

Two divide-by-N blocks are available, DIV1N and DIV2N, each with a range from 4 to 127. The source for each divider can either be the VCO or the input reference. The divider outputs are then used to generate the resulting output signal.

Outputs

Each of the six outputs can have a different source as indicated by the combobox. The choice of this source directly determines the clock frequency for that output. Each output can then be independently enabled or disabled using the checkboxes to the right.

EEPROM Read

The XEM stores the microcontroller bootcode in a small serial EEPROM which is also used to store a single set of PLL parameters. These parameters are loaded before each FPGA configuration so that valid clock signals are presented to the FPGA when it comes out of configuration.

The PLL Configuration Dialog allows you to read and write this section of EEPROM by using the buttons at the lower left. When you click the button labelled "EEPROM Read," the stored PLL configuration is read from the EEPROM and the PLL Configuration Dialog is updated to represent these values. *The PLL is not re-configured yet. To configure the PLL with these values, you must press "Apply."*

EEPROM Write

The current configuration represented in the PLL Configuration Dialog (not the current PLL configuration) is written to the EEPROM when you press this button. The next time a configuration file is downloaded to the FPGA, this configuration will be loaded into the PLL.

Apply

Any time you change a setting in the PLL Configuration Dialog or load the EEPROM settings, the values change in the dialog, but do not affect the actual PLL on-board. To make the changes take effect, you must press the "Apply" button.

Example PLL Configurations

The table below lists several example frequencies and the PLL settings required to generate that output. If more than one frequency is required for the FPGA, remember that the PLL only has a single VCO, so the outputs must be generated from a single source and (possibly) multiple divider values.

Output Frequency	P	Q	VCO Frequency	DIV1N	Source
100 MHz	400	48	400 MHz	4	DIV1CLK/DIV1N
80 MHz	20	4	240 MHz	N/A	DIV1CLK/3
75 MHz	300	48	300 MHz	N/A	DIV2CLK/4
66.66 MHz	400	48	400 MHz	6	DIV1CLK/DIV1N
50 MHz	400	48	400 MHz	8	DIV1CLK/DIV1N
15 MHz	20	4	240 MHz	16	DIV1CLK/DIV1N

Of course, many other configurations are possible including those with multiple output frequencies. Please see the specific PLL datasheet for more information.

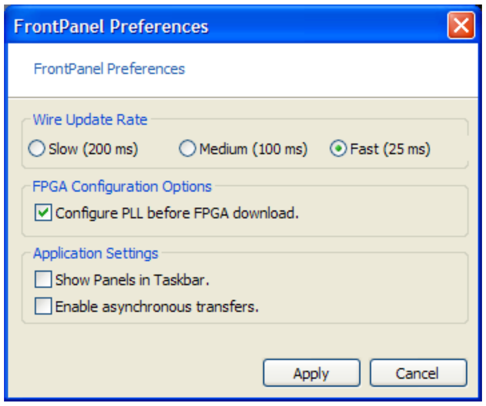
PLL Configuration (CY22393)

The XEM3010 and XEM3050 products include a Cypress CY22393 PLL which has a multi-PLL configuration and is therefore more capable than the CY22150. Configuration for the Cypress CY22393 is also available through the FrontPanel API and the FrontPanel Application. Please refer to the Cypress datasheet for parameter details on the CY22393.

Preferences

The Preferences dialog (shown below) can be shown by navigating under the FrontPanel menu:

FrontPanel → Preferences...



Wire Update Rate: Slow (200 ms) Medium (100 ms) [✓ Fast (25 ms)]

✓ Configure PLL before FPGA download.

☐ Show Panels in Taskbar.

☐ Enable asynchronous transfers.

Wire Update Rate

Wire Out enpoints are updated using timed polling by the FrontPanel software. This update rate is determined by your design's needs (how quickly you need to see wire changes) as well as the performance of your PC. On an Athlon 2100+, even the fastest update rate places minimal (<2%) load on the CPU.

Configure PLL Before FPGA Download

This option determines whether an FPGA download configures the PLL prior to download. In most cases, this is the desired behavior so that a valid clock is available when the FPGA comes out of the configuration state. Sometimes, however, you may want to keep the current PLL settings in effect and not update the EEPROM.

Show Panels in Taskbar

When unchecked, each FrontPanel "panel" is displayed in a toolbox window which does not register with the taskbar. When checked, these panels will register with the taskbar so that you can easily select a particular panel.

Enable Asynchronous Transfers (USB devices only)

Asynchronous transfers allow USB transfer requests to be queued and sequenced by the operating system. This decreases software overhead and increases overall throughput. However, many Windows 2000-based machines have problems with asynchronous transfers and may not communicate with the FPGA properly when this feature is enabled.

You may find that you need to disable asynchronous transfers before any FPGA communication. Otherwise, the communication link may become "tainted" and will not work. Therefore, if you experience problems with Windows 2000 and FrontPanel communication, we advise that you disable asynchronous transfers before communicating with your board.

From within your own software, there is an API method to control this feature.

Command Line Arguments

The FrontPanel executable may be started on the command line with some arguments to automate startup activities.

Loading a Bitfile

Use this argument to tell FrontPanel to startup, detect devices, and download the specified FPGA bitfile to the device.

```
FrontPanel.exe --load-bitfile=counters.bit
```

Loading a FrontPanel Profile

Use this argument to tell FrontPanel to startup, detect devices, and load the specified FrontPanel profile.

```
FrontPanel.exe --load-profile=counters.xfp
```

Selecting a Device by Serial Number

By default, FrontPanel will apply command line arguments to the first device detected. You can optionally specify a device for subsequent command line arguments with this option.

```
FrontPanel.exe --device-serial=12340009819 \
               --load-bitfile=counters.bit
               --load-profile=counters.xfp
```

Component XML

FrontPanel user interfaces ("panels") are constructed from "components" - graphical devices that interface to your design or serve some decorative function. The interfaces are described in FrontPanel "profiles" which are written in a text file format known as XML. The XML profile contains structure which defines where each component exists on a panel as well as the connections that component has to your FPGA design. FrontPanel XML files end with the extension XFP.

XML

XML stands for eXtensible Markup Language and is used in documents containing structured information. The syntax for XML is defined at http://www.w3.org/TR/WD-xml. For its part, XML does not define the content but rather how the content is organized. FrontPanel uses XML because the standard is well-known and there are many tools available to read, write, edit, and parse the content. It is also easily human-readable so you can read and write FrontPanel profiles in a text-editor with ease.

A complete tutorial of XML is beyond the scope of this text. What is provided here is a basic tutorial of the aspects of XML required to compose FrontPanel profiles. Please refer to the enormous on-line resources available for a complete understanding of XML, its applications, and the tools available for working with XML.

Basic Structure for FrontPanel

The basic FrontPanel XFP file has the following structure:

```
<?xml version=”1.0” encoding=”ISO-8859-1”?>
<!-- A basic FrontPanel Example -->

<resource version=”2.3.0.1”>
<object class=”okPanel” name=”panel1”>
    <title>Main Panel Title</title>
    <size>180,70</size>

    ... Main Panel component XML will go here ...

</object>
</resource>
```

This simple example defines a single panel. Note the first line starting with <?xml...> and the <resource version=”2.3.0.1”> ... </resource> are required content in any FrontPanel profile.

Comments

Comments in XML can appear anywhere outside of normal markup. They have the form as shown below. Note that the string “--” is not allowed within a comment and that the comment must end with exactly two “-” (hyphen) characters and the “>” character.

```
<!-- This is some comment text.  -->

<!-- This text is NOT allowed because it is incorrectly terminated: --->
```

Start-Tags and End-Tags

The start- and end-tags enclose an XML element. In the listing below, the XML element is an “object” and its content is defined between the first line (Start-Tag) and last line (End-Tag). This particular element contains a child element, “label” which also has (as a requirement) a Start-Tag and an End-Tag.

```
<object class=”okStaticText”>
   <label>Hello there</label>
</object>
```

The “object” element in the above example contains one attribute, “class” which is set to “okStaticText”. In FrontPanel, all of the graphical components are “object” elements with an attribute which defines what type of component it is.

Case Sensitivity

All XML component types and value names are currently case sensitive. That is, “okPushButton” is not a valid component name, but “okPushbutton” is.

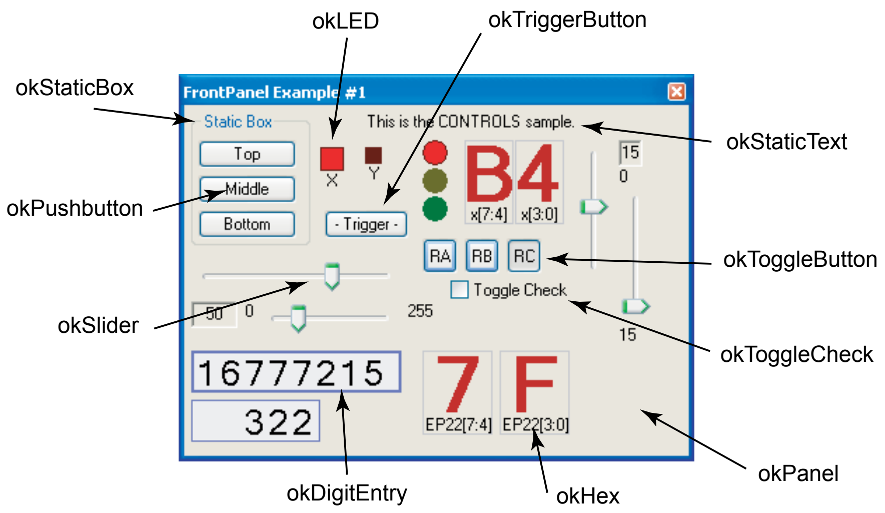
Element Data Types

All FrontPanel components have sub-elements which specify certain properties of the component. These sub-elements are listed with each component and take a certain data type as their value. The various data types available along with an example and description are shown in the table below.

Type	Example	Description
POSITION	50,75	Position represented as: x,y in pixels.
SIZE	40,80	Size represented as: width,height in pixels. Many controls will accept -1 as a width and/or height and automatically compute the best value.
TEXT	Hello World	A text string. No quotes are necessary.
HEX BYTE	0x3F	An 8-bit hexadecimal number. The leading "0x" is required.
NUMBER	7	Numeral, range is determined by object type.
BIGNUMBER	179 0x7FFFFFF	Like NUMBER, but also supports hexadecimal with a "0x" prefix. Decimal values are supported to 31 bits. Hexadecimal values are supported to 63 bits.
COLOR	#2040A3	24-bit hexadecimal HTML color format #RRGGBB.
STYLE	ROUND	The STYLE type is object-dependent and contains one or more styles which can be or'ed together using the pipe ("|") symbol.

Component Types

The following figure shows most of the FrontPanel components available. Some components do not have a corresponding GUI representation. This image is taken from the Controls Sample.



okStaticText

This is a simple control to display static text within a panel.

Element	Type	Description
position	POSITION	Position of the top left corner.
size	SIZE	Size in pixels.
label	TEXT	Label text.

XML Example:

```
<object class=”okStaticText”>
    <label>Disable</label>
    <position>90,25</position>
    <size>60,20</size>
</object>
```

okStaticBox

This is a simple control to display static text within a panel. It also displays a box which is helpful to distinguish parts of a control panel.

Element	Type	Description
position	POSITION	Position of the top left corner.
size	SIZE	Size in pixels.
label	TEXT	Label text.

XML Example:

```
<object class=”okStaticBox”>
    <label>Disable</label>
    <position>90,25</position>
    <size>60,20</size>
</object>
```

okPushbutton (Wire In)

This component models a physical pushbutton and connects to a Wire In endpoint. By default, the pushbutton is ‘unpressed’ and the corresponding wire is deasserted (logic 0). When pressed, the corresponding wire is asserted (logic 1). The pushbutton does not hold its state -- that is, to maintain a logic 1, you have to hold the pushbutton in its pressed state.

For an alternative component that does maintain its state, see the okToggleButton.

Element	Type	Description
position	POSITION	Position of the top left corner.
size	SIZE	Size in pixels.
label	TEXT	Label text, shown inside the button.
tooltip	TEXT	Tooltip text.
endpoint	HEX BYTE	Endpoint address for the corresponding Wire In.
bit	NUMBER	Bit to which this component addresses (0=LSB, 15=MSB).

XML Example:

```
<object class="okPushbutton">
    <label>Disable</label>
    <position>90,25</position>
    <size>60,20</size>
    <endpoint>0x00</endpoint>
    <bit>1</bit>
    <tooltip>Momentarily disable counter #1</tooltip>
</object>
```

okToggleButton (Wire In)

The okToggleButton is similar to the okPushbutton in that it connects to a Wire In endpoint. In contrast to the okPushbutton, however, this component maintains its state just as a physical toggle switch would. When unpressed, the corresponding wire is deasserted (logic 0). When pressed, the corresponding wire is asserted (logic 1).

Note: This component is not presently available under OS X.

Element	Type	Description
position	POSITION	Position of the top left corner.
size	SIZE	Size in pixels.
label	TEXT	Label text, shown inside the button.
tooltip	TEXT	Tooltip text.
endpoint	HEX BYTE	Endpoint address for the corresponding Wire In.
bit	NUMBER	Bit to which this component addresses (0=LSB, 15=MSB).

XML Example:

```
<object class="okToggleButton">
    <label>1</label>
    <position>10,10</position>
    <size>20,20</size>
    <endpoint>0x00</endpoint>
    <bit>0</bit>
</object>
```

okToggleCheck (Wire In)

The okToggleCheck attaches to a Wire In component and behaves much like the okToggleButton except that graphically it appears as a checkbox with the label text on the right. When un-checked, the corresponding wire is unasserted (logic 0). When checked, the corresponding wire is asserted (logic 1).

Element	Type	Description
position	POSITION	Position of the top left corner.
size	SIZE	Size in pixels. If no size is specified, the component is automatically sized.
label	TEXT	Label text, shown inside the button.
tooltip	TEXT	Tooltip text.
endpoint	HEX BYTE	Endpoint address for the corresponding Wire In.
bit	NUMBER	Bit to which this component addresses (0=LSB, 15=MSB).

XML Example:

```
<object class="okToggleCheck">
    <label>Autocount.</label>
    <position>20,135</position>
    <endpoint>0x00</endpoint>
    <bit>2</bit>
    <tooltip>Enable autocount.</tooltip>
</object>
```

okDigitEntry (Wire In)

This component allows a more flexible way to convey numerical information to your design. The okDigitEntry attaches to one or more Wire In endpoints and allows the user to enter a numerical value using the mouse and/or keyboard. The bounds on the value are set in the component properties.

The okDigitEntry component is designed to allow fast entry through either the mouse or keyboard. Using the mouse, you can hover over any digit and change its value using the scrollwheel. Likewise, by pressing a number on the keyboard when a digit is highlighted, that particular digit is changed and the highlight moves to the next digit on the right.

Element	**Type**	**Description**
position	POSITION	Position of the top left corner.
size	SIZE	Size in pixels.
tooltip	TEXT	Tooltip text.
endpoint	HEX BYTE	Endpoint address for the corresponding Wire In. The entry will span multiple consecutive endpoints as necessary.
bit	NUMBER	This bit on the endpoint is the LSB for the entry.
minvalue	NUMBER	The minimum allowed value in the entry.
maxvalue	BIGNUMBER	The maximum allowed value in the entry.
raidx	NUMBER	Numerical radix of the entry (2, 8, 10 [default], or 16).
value	NUMBER	The default value for the entry.

XML Example:

```
<object class="okDigitEntry">
    <position>5,215</position>
    <size>200,30</size>
    <tooltip>Sets the integer divider.</tooltip>
    <minvalue>0</minvalue>
    <maxvalue>16777215</maxvalue>
    <value>49837</value>
    <endpoint>0x07</endpoint>
    <bit>0</bit>
</object>
```

okSlider (Wire In)

Element	Type	Description
position	POSITION	Position of the top left corner.
size	SIZE	Size in pixels.
label	TEXT	Label text, shown inside the button.
tooltip	TEXT	Tooltip text.
endpoint	HEX BYTE	Endpoint address for the corresponding Wire In.
bit	NUMBER	Bit to which this component addresses (0=LSB, 15=MSB).
minvalue	NUMBER	The minimum value on the slider.
maxvalue	NUMBER	The maximum value on the slider.
value	NUMBER	Default value taken when the profile is loaded.
style	STYLE	VERTICAL - Displays the slider vertically. HORIZONTAL - Displays the silder horizontally. SHOWLABELS - Show min/max/value labels.

XML Example:

```
<object class="okSlider">
    <position>310,5</position>
    <size>25,100</size>
    <label>Hi</label>
    <tooltip>4-bit vertical slider.</tooltip>
    <style>VERTICAL|SHOWLABELS</style>
    <minvalue>0</minvalue>
    <maxvalue>15</maxvalue>
    <value>3</value>
    <endpoint>0x04</endpoint>
    <bit>4</bit>
</object>
```

okCombobox (Wire In)

The okCombobox allows you to relate numerical values on a Wire In endpoint to text selections in a traditional combobox. You specify the text items and a corresponding value. When that text item is selected, the Wire In is updated with the value.

Element	Type	Description
position	POSITION	Position of the top left corner.
size	SIZE	Size in pixels.
tooltip	TEXT	Tooltip text.
endpoint	HEX BYTE	Endpoint address for the corresponding Wire In.
bit	NUMBER	Bit to which this component addresses (0=LSB, 15=MSB).
options	XML	This field is further broken down into 'item' tags as shown in the example below. Each item tag is inserted in order into the combobox. Each item tag has a 'value' property which specifies the Wire In value to be used for each item selection.

XML Example:

```
<object class=”okCombobox”>
    <position>180,160</position>
    <size>100,-1</size>
    <options>
        <item value=”0”>Test mode</item>
        <item value=”1”>Standard mode</item>
        <item value=”2”>Block floating point mode</item>
    </options>
    <endpoint>0x01</endpoint>
    <bit>1</bit>
</object>
```

okLED (Wire Out)

This component implements a simple on/off indicator analagous to a physical LED. It is attached to a specified bit on a specified Wire Out endpoint and monitors the status of that bit. Both the style (round or square) and color (a 24-bit RGB value) may be specified.

The LED is on when the Wire Out is asserted (logic 1) and off when the Wire Out is deasserted (logic 0). When on, the LED is displayed in the specified color. When off, the LED is darkened.

Element	Type	Description
position	POSITION	Position of the top left corner.
size	SIZE	Size in pixels.
label	TEXT	Label text. The optional ‘align’ property can be “left | right | top | bottom” and specifies the text alignment relative to the LED.
tooltip	TEXT	Tooltip text.
endpoint	HEX BYTE	Endpoint address for the corresponding Wire Out.
bit	NUMBER	Bit to which this component addresses (0=LSB, 15=MSB).
color	COLOR	The LED “on” color. The “off” color is automatically computed as a darker version of this color.
style	STYLE	ROUND - Displays a round LED. SQUARE - Displays a square LED.

XML Example:

```
<object class=”okLED”>
    <position>135,50</position>
    <size>25,25</size>
    <label align=”top”>1</label>
    <style>SQUARE</style>
    <color>#00ff00</color>
    <endpoint>0x20</endpoint>
    <bit>1</bit>
</object>
```

okHex (Wire Out)

The okHex component displays four bits of a Wire Out endpoint as a hexadecimal digit. Multiple okHex components may be attached to the same Wire Out endpoint. For example, to display an entire byte in hex, you could display two okHex components side-by-side. Attach the left component to bit 4 and the right component to bit 0.

Element	Type	Description
position	POSITION	Position of the top left corner.
size	SIZE	Size in pixels.
label	TEXT	Label text, shown inside the button.
tooltip	TEXT	Tooltip text.
endpoint	HEX BYTE	Endpoint address for the corresponding Wire Out.
bit	NUMBER	Least-significant bit to which this component addresses. The hex value comes from the specified bit and its three neighbors to the left. For example, if bit=2, the hex value will be taken from bits 5:2.
color	COLOR	Sets the numeral color.

XML Example:

```
<object class=”okHex”>
    <label>x[3:0]</label>
    <position>217,22</position>
    <size>35,50</size>
    <endpoint>0x20</endpoint>
    <bit>0</bit>
    <tooltip>Counter #1 (low nibble)</tooltip>
</object>
```

okDigitDisplay (Wire Out)

This component allows a flexible way to display numerical information to your design. The okDigitDisplay is simply a read-only (Wire Out) version of the okDigitEntry. Just like the okDigitEntry, its endpoint attachment can span multiple Wire Out endpoints as necessary (according to the 'maxvalue' setting).

Element	Type	Description
position	POSITION	Position of the top left corner.
size	SIZE	Size in pixels.
label	TEXT	Label text, shown inside the button.
tooltip	TEXT	Tooltip text.
endpoint	HEX BYTE	Endpoint address for the corresponding Wire Out. The display will span multiple consecutive endpoints as necessary.
bit	NUMBER	This bit on the endpoint is the LSB for the display.
maxvalue	BIGNUMBER	The maximum allowed value in the display.
radix	NUMBER	Numerical radix of the entry (2, 8, 10 [default], or 16).

XML Example:

```
<object class=”okDigitDisplay”>
    <position>5,215</position>
    <size>200,30</size>
    <maxvalue>65535</maxvalue>
    <radix>16</radix>
    <endpoint>0x20</endpoint>
    <bit>0</bit>
</object>
```

okGauge (Wire Out)

The okGauge component is used to display a bar-type indicator horizontally or vertically on the panel. It connects to a Wire Out endpoint and allows a maximum range of 65535 (all 16-bits of a Wire Out). It appropriately selects the proper number of bits for smaller ranges.

Element	Type	Description
position	POSITION	Position of the top-left corner.
size	SIZE	Size in pixels.
tooltip	TEXT	Tooltip text.
style	TEXT	Either "HORIZONTAL" or "VERTICAL"
endpoint	HEX BYTE	Endpoint address for the corresponding Wire Out.
bit	NUMBER	This bit on the endpoint is the LSB for the display.
range	NUMBER	The maximum allowed value in the display.

XML Example:

```
<object class=”okGauge”>
    <position>120,235</position>
    <size>150,15</size>
    <style>HORIZONTAL</style>
    <range>65535</range>
    <endpoint>0x33</endpoint>
    <bit>0</bit>
</object>
```

okTriggerButton (Trigger In)

The okTriggerButton appears identical to the okPushbutton but connects to a Trigger In endpoint. The trigger is activated when the button is pushed (rather than when the button is released).

You may wish to denote that a particular button is a trigger by surrounding the label with hyphens. In the example below, the button label is "- Reset -" to make the button appear different from an okPushbutton.

Element	Type	Description
position	POSITION	Position of the top-left corner.
size	SIZE	Size in pixels.
label	TEXT	Label text, shown inside the button.
tooltip	TEXT	Tooltip text.
endpoint	HEX BYTE	Endpoint address for the corresponding Trigger In.
bit	NUMBER	Bit to which this component addresses (0=LSB, 15=MSB).

XML Example:

```
<object class=”okTriggerButton”>
    <label>- Reset -</label>
    <position>20,110</position>
    <size>60,20</size>
    <endpoint>0x40</endpoint>
    <bit>0</bit>
    <tooltip>Reset Counter #2</tooltip>
</object>
```

okTriggerSound (Trigger Out)

The okTriggerSound does not physically appear on a virtual panel. Instead, it is attached to the panel and is activated when a trigger out is activated. Upon activation, it rings the system bell as a brief audible notification of a trigger out event. An optional WAV file may be specified that will play instead of the system bell.

Element	Type	Description
endpoint	HEX BYTE	Endpoint address for the corresponding Trigger In.
bit	NUMBER	Bit to which this component addresses (0=LSB, 15=MSB).
label	TEXT	Label text, shown in the FrontPanel component list. (OPTIONAL)
soundfile	FILENAME	Filename of a WAV file to be played upon triggering. (OPTIONAL)

XML Example:

```
<object class=”okTriggerSound”>
    <endpoint>0x63</endpoint>
    <bit>3</bit>
    <label>Transfer complete trigger.</label>
    <soundfile>c:/Windows/Media/chimes.wav</soundfile>
</object>
```

okTriggerLog (Trigger Out)

okTriggerLog displays specified Trigger Out events along with a user-specified text message in list form. Each trigger item within the list is stamped with the time (hh:mm:ss) of the occurrance.

Element	Type	Description
position	POSITION	Position of the top-left corner.
size	SIZE	Size in pixels.
trigger	XML	Adds a message to be entered in the log when a trigger event occurs. The XML contains the ‘endpoint’, ‘bit’, and ‘message’ tags as shown in the example below.

XML Example:

```
<object class=”okTriggerLog”>
    <position>5,290</position>
    <size>350,100</size>
    <trigger>
        <endpoint>0x60</endpoint><bit>1</bit>
        <message>Your laundry is done.</message>
    </trigger>
    <trigger>
        <endpoint>0x61</endpoint><bit>0</bit>
        <message>Elvis (the cat) has left the building.</message>
    </trigger>
</object>
```

okTriggerMessage (Trigger Out)

okTriggerMessage displays a brief text message, similar to an okStaticText display, when a particular trigger occurs. Similar to okTriggerLog, you can setup a variety of messages to be displayed in the same area when any of a number of trigger outs occur.

Element	Type	Description
position	POSITION	Position of the top-left corner.
size	SIZE	Size in pixels.
style	STYLE	Acceptable border styles are: (none means no border) SIMPLE_BORDER RAISED_BORDER SUNKEN_BORDER Acceptable text styles are: ALIGN_LEFT (default) ALIGN_RIGHT ALIGN_CENTER
trigger	XML	Adds a message to be displayed when a trigger event occurs. The XML contains 'endpoint', 'bit', 'delay', and 'message' tags as shown in the example below. The 'delay' parameter specifies an optional delay (in seconds), after which the message will disappear.

XML Example:

```
<object class="okTriggerMessage">
    <position>5,290</position>
    <size>200,20</size>
    <style>RAISED_BORDER|ALIGN_CENTER</style>
    <trigger>
        <endpoint>0x60</endpoint><bit>1</bit>
        <message>Your laundry is done.</message>
        <delay>0.5</delay>
        <background>#ff0000</background>
        <foreground>#ffffff</foreground>
    </trigger>
    <trigger>
        <endpoint>0x61</endpoint><bit>0</bit>
        <message>Elvis (the cat) has left the building.</message>
    </trigger>
</object>
```

okFilePipe (Pipe In, Pipe Out, Trigger In)

This component provides simple binary file transfer capability through the use of Pipe In or Pipe Outs. The type (In or Out) is automatically determined by the endpoint address. The component appears as a pushbutton on your panel that can be clicked to initiate the transfer.

If no filename is provided, the user will be prompted with a File Dialog to select an appropriate input or output file. If a filename is provided for Pipe In, but the file does not exist, the user will also be prompted.

In the case of a Pipe In, the filename parameter provides an input file. The entire contents of the file are transferred to the Pipe In. The transfer proceeds in chunks of 64kB until the entire file has been transferred.

In the case of a Pipe Out, a length parameter must be provided to tell FrontPanel how many bytes to read from the FPGA. The transfer proceeds in chunks of 64kB until the full length has been read and stored.

In both cases, an optional Start Trigger and optional Done Trigger are available. The Start Trigger will be activated just before the transfer initiates. The Done Trigger is activated after the transfer completes. These triggers can be used as notification events within your hardware.

To use a BlockPipeIn or BlockPipeOut, specify the blocksize parameter appropriately. Please refer to the FrontPanel API Reference for blocksize limitations depending on the interface type.

Element	Type	Description
position	POSITION	Position of the top left corner.
size	SIZE	Size in pixels.
label	TEXT	Label text, shown inside the button.
tooltip	TEXT	Tooltip text.
endpoint	HEX BYTE	Endpoint address for the corresponding Pipe In or Pipe Out.
filename	TEXT	Optional filename to read or write. If not provided, the user will be prompted.
length	NUMBER	For Pipe Out transfers, the length (in bytes) to read from the Pipe Out and store in the file.
blocksize	NUMBER	Optionally specifies a block size to be used for Block Pipes. If unspecified, standard Pipes will be used. If specified, Block Pipes will be used.
append		If present, an output file will be appended if it already exists.
starttrigger	XML	Describes the parameters of an optional Start Trigger (endpoint and bit).
donetrigger	XML	Describes the parameters of an optional Done Trigger (endpoint and bit).

XML Example:

```
<object class=”okFilePipe”>
    <label>Pipe Out</label>
    <position>20,53</position>
    <size>60,20</size>
    <endpoint>0xa0</endpoint>
    <length>5000</length>
    <tooltip>Read a file from Pipe 0xA0</tooltip>
    <append />
    <starttrigger><endpoint>0x40</endpoint><bit>0</bit></starttrigger>
    <donetrigger><endpoint>0x40</endpoint><bit>1</bit></donetrigger>
</object>
```

okPLL22150

This component provides an XML method to program the on-board PLL. When provided with a “label” parameter, this component becomes a pushbutton on the panel GUI. When that button is pressed, the PLL is configured with the given parameters. This allows you to specify multiple PLL configuration and provide multiple buttons to access them without going through the PLL dialog. A convenient tooltip lists the VCO and output frequencies for the configuration.

When the component does not have the “label” parameter, this configuration is loaded to the PLL when the profile is loaded. It is not stored to EEPROM and the component does not create a GUI button. The only way to reconfigure the PLL (even after a new FPGA configuration file is loaded) is to reload the profile.

Note that this element is ignored if the target device does not have a CY22150 PLL.

Element	Type	Description
position	POSITION	Position of the top left corner. (OPTIONAL)
size	SIZE	Size in pixels. (OPTIONAL)
label	TEXT	Label text, shown inside the button. (OPTIONAL)
p	NUMBER	VCO P multiplier. [8..2055]
q	NUMBER	VCO Q divider. [2..129]
divider1 divider2	NUMBER	Divider 1 N value. [4..127] The parameter "source" is a string that represents the source of the divider: "ref" - The reference (48 MHz) is used. "vco" - The VCO frequency (48 * P / Q) is used.
output0 output1 ... output5	STRING	This string is either "on" or "off" and turns the output on or off. The parameter "source" is a string that represents the source for the output: "ref" - Use the reference (48 MHz). "div1byn" - Use divider 1 source divided by divider 1 N. "div1by2" - Use divider 1 source divided by 2. "div1by3" - Use divider 1 source divided by 3. "div2byn" - Use divider 2 source divided by divider 2 N. "div2by2" - Use divider 2 source divided by 2. "div2by4" - Use divider 2 source divided by 4.

XML Example:

```
<object class=”okPLL22150”>
    <label>PLL1 Configuration</label>
    <position>170,5</position>
    <size>100,15</size>
    <p>400</p>
    <q>48></q>
    <divider1 source=”vco”>8</divider1>
    <output0 source=”div1byn”>on</output0>
</object>
```

okPLL22393

This component provides an XML method to program the on-board PLL. When provided with a "label" parameter, this component becomes a pushbutton on the panel GUI. When that button is pressed, the PLL is configured with the given parameters. This allows you to specify multiple PLL configuration and provide multiple buttons to access them without going through the PLL dialog. A convenient tooltip lists the VCO and output frequencies for the configuration.

When the component does not have the "label" parameter, this configuration is loaded to the PLL when the profile is loaded. It is not stored to EEPROM and the component does not create a GUI button. The only way to reconfigure the PLL (even after a new FPGA configuration file is loaded) is to reload the profile.

Note that this element is ignored if the target device does not have a CY22393 PLL. Also note that the convention here is to label PLLs and outputs as 0-indexed (0, 1, 2, ...) rather than indexed from 1 as the Cypress documentation does.

Element	Type	Description
position	POSITION	Position of the top left corner. (OPTIONAL)
size	SIZE	Size in pixels. (OPTIONAL)
label	TEXT	Label text, shown inside the button. (OPTIONAL)
pll0 pll1 pll2	none	This parameter has no content, but does have the following properties: P - Specifies the P multiplier for the PLL. [6..2053] Q - Specifies the Q divider for the PLL. [2.257]
output0 output1 ... output4	STRING	This string is either "on" or "off" and turns the output on or off. The property "source" is a string that represents the source for the output: "ref" - Use the reference (48 MHz). "pll0_0" - PLL ouput 0 with 0˚ phase shift. "pll0_180" - PLL ouput 0 with 180˚ phase shift. "pll1_0" - PLL ouput 1 with 0˚ phase shift. "pll1_180" - PLL ouput 1 with 180˚ phase shift. "pll2_0" - PLL ouput 2 with 0˚ phase shift. "pll2_180" - PLL ouput 2 with 180˚ phase shift. The property "divider" specifies the integer divider for the output. [1..127] for outputs 0..3 and [2,3,4] for output 4.

XML Example:

```
<object class=”okPLL22393”>
    <label>PLL1 Configuration</label>
    <position>170,5</position>
    <size>100,15</size>
    <pll0 p=”400” q=”48”/>
    <pll1 p=”397” q=”43”/>
    <output0 source=”pll0_0” divider=”8”>on</output0>
    <output1 source=”pll1_180” divider=”16”>on</output0>
</object>
```

okKeyPanel (Wire In, Trigger In)

The okKeyPanel component allows keyboard input to be captured and mapped to selected Wire In and Trigger In endpoints. Multiple okKeyPanels may be instantiated on the same okPanel allowing the same keyboard events to map to different behaviors depending on which okKeyPanel is active.

The okKeyPanel appears on a panel as a simple box with a text label within. When the mouse is over the component, it changes color to indicate that it is active. When active, keyboard events are captured and mapped to HDL endpoints according to the XML description. Three behaviors are available: KeyButton, KeyToggle, and KeyTrigger.

KeyButton

The KeyButton works like a pushbutton. The Wire In is asserted when the key is pressed and deasserted when the key is released.

KeyToggle

The KeyButton is like a toggle button. On the key downstroke, the Wire In is toggled. Nothing happens on the upstroke.

KeyTrigger

The KeyTrigger activates a Trigger In when the keyboard event occurs. By default, the keyboard event is defined as the key downstroke. However, with the optional <up/> tag within the XML, the KeyTrigger can map to the upstroke. By defining both the upstroke and downstroke to the same key, triggers can be sent on each end of a keypress.

Element	Type	Description
position	POSITION	Position of the top left corner.
size	SIZE	Size in pixels.
label	TEXT	Label text.
color	COLOR	Sets the component's active color.
keys	XML	XML describing the key mapping from keyboard events to HDL endpoints. See the table below for more details.

The following table describes the nodes of the <key> XML element within the component description. This mapping is used to associate a keyboard event with an HDL endpoint.

Element	Type	Description
KeyButton	XML	Defines a KeyButton behavior on the provided keycode to the associated Wire In endpoint. The "keycode" property defines the mapped key.
KeyToggle	XML	Defines a KeyTrigger behavior on the provided keycode to the associated Wire In endpoint. The "keycode" property defines the mapped key.
KeyTrigger	XML	Defines a KeyTrigger behavior on the provided keycode to the associated Trigger In endpoint. The "keycode" property defines the mapped key.

The table below lists the recognized keycodes.

KEY_A ... KEY_Z	KEY_UP	KEY_NUMPAD0 ... KEY_NUMPAD9
KEY_0 ... KEY_9	KEY_DOWN	KEY_NUMLOCK
KEY_F1 ... KEY_F24	KEY_LEFT	KEY_NUMPADDIV
KEY_BACK	KEY_RIGHT	KEY_NUMPADMULT
KEY_TAB	KEY_INSERT	KEY_NUMPADADD
KEY_RETURN	KEY_DELETE	KEY_NUMPADSUB
KEY_ESCAPE	KEY_END	KEY_NUMPADDECIMAL
KEY_SPACE	KEY_HOME	
KEY_SHIFT	KEY_PGUP	
KEY_CONTROL	KEY_PGDOWN	

XML Example:

```
<object class=”okKeyPanel”>
    <label>Key Panel A</label>
    <color>#b0f0b0</color>
    <position>5,260</position>
    <size>100,55</size>
    <keys>
        <KeyButton keycode=”KEY_UP”>
            <endpoint>0x00</endpoint><bit>0</bit>
        </KeyButton>
        <KeyToggle keycode=”KEY_DOWN”>
            <endpoint>0x00</endpoint><bit>1</bit>
        </KeyButton>
        <KeyTrigger keycode=”KEY_A”>
            <endpoint>0x40</endpoint><bit>1</bit>
        </KeyTrigger>
        <KeyTrigger keycode=”KEY_A”>
            <up/>
            <endpoint>0x40</endpoint><bit>1</bit>
        </KeyTrigger>
    </keys>
</object>
```

FrontPanel Host Simulation

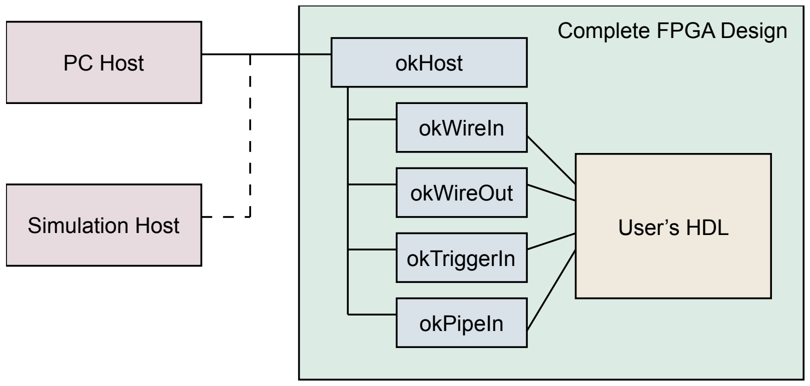
Hardware simulation is a valuable tool used to reduce design cycles and quickly debug a hardware design. While debug outputs to real instruments (logic analyzers and oscilloscopes) as well as the virtual instruments supported by FrontPanel can help in the controllability and observability of a design, nothing can match the flexibility offered by simulation.

Unfortunately, full system simulation is often difficult to attain. Simulation models of external hardware are often not available. More importantly, integration of the hardware simulation with software can be difficult.

The FrontPanel API provides a simple, capable, and convenient communication interface between the hardware design residing within the FPGA and a user application running on a PC host. The Opal Kelly FrontPanel Host Simulation Libraries allow simulation of this PC host within a hardware simulation.

System Simulation Model

The block diagram below illustrates the system simulation model for the Host Simulation Libraries. The FPGA design encompasses the user's HDL design as well as the okHost module and endpoint modules (such as okWireIn and okPipeOut). In a live system, the okHost communicates with the USB microcontroller on the FPGA board which, in turn, communicates with the PC and software API. In the simulation system, the okHost is replaced by a simulation model which communicates with a simulation model for the Host. The user's simulation test fixture executes host directives as if they were software API calls.



The goal of this type of simulation model is to simulate the complete FPGA design without having to make changes specific to the simulation model. In reality, many designs will require some modification, but in this case the host can be simulated as realistically as possible.

Simulation Requirements

The Opal Kelly FrontPanel Host Simulation Libraries are provided as source HDL to allow for compatibility with a wide range of simulation packages. Examples are provided below for Modelsim and iSim packages packaged with the Xilinx ISE and Altera Quartus toolsets. Verilog and VHDL source is located in the directories under the Simulation subdirectories of the FrontPanel installation location for compilation with the users test fixture.

Limitations

Opal Kelly's FrontPanel Host Simulation library is for behavioral simulation only. Post place & route simulation is not supported.

Test Fixture Simulation Requirements

A test fixture which simulates the FrontPanel Host requires the following components:

1. Instantiation of the device under test (DUT). This is required in any test fixture.
2. A behavioral block which calls the Host Simulation Library to mimic the FrontPanel API.
3. Inclusion of `okHostCalls` to simulate the various host API functions.
   Verilog: Include `okHostCalls.v` in the test fixture with `‘include “okHostCalls.v”`
   VHDL: User must copy indicated code segment from `okHostCalls_vhd.txt` into the test fixture process.

The last two items are specific to FrontPanel Host Simulation. The table below lists the FrontPanel API calls that are available within the Host Simulation Library. In most cases, the parameters are identical to the corresponding FrontPanel API calls.

SetWireIns	GetWireOutValue
ActivateTriggerIn	IsTriggered
WriteToPipeIn	ReadFromPipeOut
WriteToBlockPipeIn	ReadFromBlockPipeOut
WriteRegister (USB 3.0)	ReadRegister (USB 3.0)

UpdateWireIns	UpdateWireOuts
UpdateTriggerOuts	FrontPanelReset

Reset

In a live FPGA design, the FPGA automatically performs a reset of all logic within the fabric after configuration. This assures that the entire design start in a known state which is established by the design.

In a simulation environment, this reset signal is not always simulated and some signals may start in an unknown state. The `FrontPanelReset` call will reset the host interface functions and assure that the simulation starts off in a known state. It is therefore recommended that your simulation issue a call `Reset` at the beginning of the simulation.

Simulating Pipes

Pipe transfer calls utilize global array variables in the test fixture to store the data that will be transmitted or received. These global variables must be declared within the user's testbench if any pipe functionality is to be simulated. In addition, the three parameters `BlockDelayStates`, `ReadyCheckDelay`, and `PostReadyDelay` determine how many clock periods exist between various pipe functions to help simulate possible delays that may occur in actual hardware. `BlockDelayStates` adds delay between transfers of blocks of data, `ReadyCheckDelay` simulates a lag in clocks before a Block Pipe module checks for a valid EP_READY signal, and `PostReadyDelay` simulates a delay after EP_READY is asserted before the next block of data is piped.

An example setup for these requirements is shown here:

```
parameter BlockDelayStates = 5;  // REQUIRED: # of clocks between blocks of pipe data
parameter ReadyCheckDelay = 5;   // REQUIRED: # of clocks before block transfer before
                                 //    host interface checks for ready (0-255)
parameter PostReadyDelay = 5;    // REQUIRED: # of clocks after ready is asserted and
                                 //    check that the block transfer begins (0-255)
parameter pipeInSize = 16383;    // REQUIRED: byte (must be even) length of default
                                 //    PipeIn; Integer 0-2^32
parameter pipeOutSize = 16383;   // REQUIRED: byte (must be even) length of default
                                 //    PipeOut; Integer 0-2^32
reg   [7:0] pipeIn [0:(pipeInSize-1)];
reg   [7:0] pipeOut [0:(pipeOutSize-1)];
```

After a call to `ReadFromPipeOut` or `ReadFromBlockPipeOut` the received data will be in the byte-wide register array `pipeOut`, arranged as it would be after a call to the C++ method. Similarly, before a call to `WriteToPipeIn` or `WriteToBlockPipeIn` the transmitted data should be setup in the byte-wide register array `pipeIn`. More pipe data arrays may be added as needed by copying and modifying the default pipe functions.

Simulation Sample

A simulation sample is included with FrontPanel to help get you started. The sample include a hypothetical FPGA design with a pseudo-random sequence generator (PRSG) with some parameters under control from the host PC.

Required Files

The following table lists the files required for the simulation along with a brief description.

Filename	Description
`sim.[v|vhd]`	This is the source HDL for the simulation example (DUT).
`sim_tf.[v|vhd]`	This is the test fixture HDL for the simulation.
`sim.do`	This files contains the ModelSim commands to setup, compile, and run the simulation. (Required for Modelsim only)
`sim_isim.bat`	This is the iSim batch file to setup, compile, and run the simulation. (Required for iSim only)
`sim_isim.prj`	iSim project file. Lists source files for iSim simulation. (Required for iSim only)
`sim_isim.tcl`	iSim command script for waveform setup. (Required for iSim only)

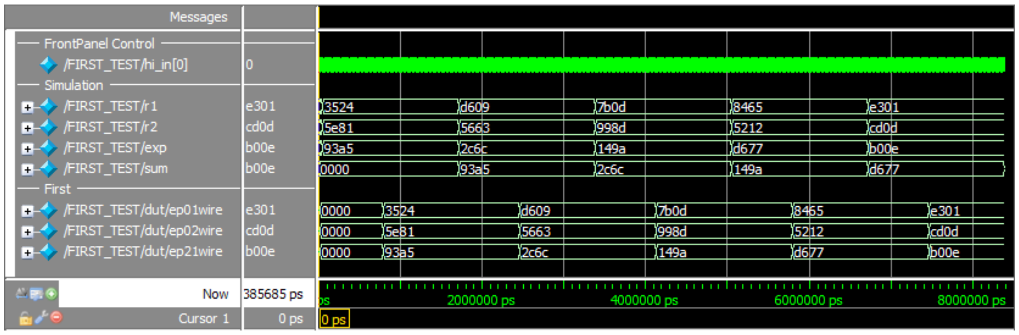
Running the Simulation

1. Copy project simulation files above from the FrontPanel installation directory `$(FRONTPANEL)/Samples/Simulation/USB[2|3]/[Verilog|VHDL]` to a work directory `$(WORKDIRECTORY)`.
2. Copy the simulation models frlom `$(FRONTPANEL)/Simulation/USB[2|3]/[Verilog|VHDL]` to `$WORKDIRECTORY/oksim`

Modelsim

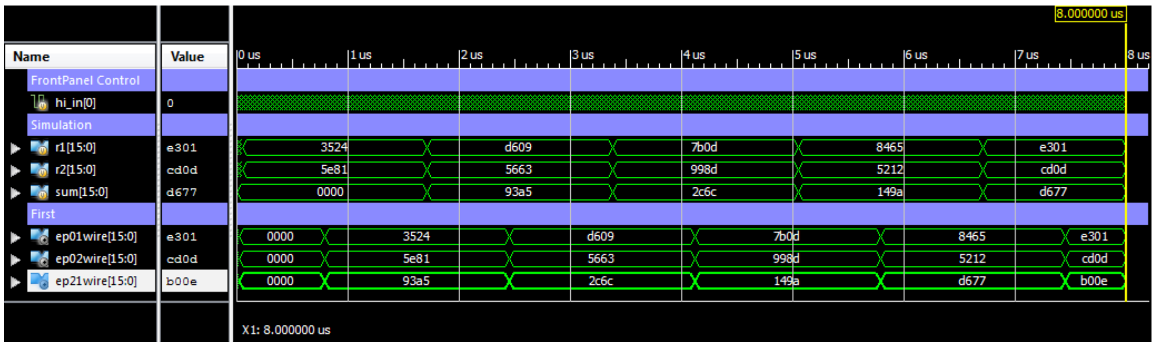
1. Start Modelsim.
2. In the Transcript window, CD to your WORK DIRECTORY:
   `cd YOUR_WORK_DIRECTORY`
3. Execute the simulation script:
   `do sim.do`

The simulation should run to completion. By selecting the "Wave" window, you should see something like this in Modelsim:


Messages
FrontPanel Control
/FIRST_TEST/hi_in[0] 0
Simulation
/FIRST_TEST/r1 e301 3524 d609 7b0d 8465 e301
/FIRST_TEST/r2 cd0d 5e81 5663 998d 5212 cd0d
/FIRST_TEST/exp b00e 93a5 2c6c 149a d677 b00e
/FIRST_TEST/sum b00e 0000 93a5 2c6c 149a d677
First
/FIRST_TEST/dut/ep01wire e301 0000 3524 d609 7b0d 8465 e301
/FIRST_TEST/dut/ep02wire cd0d 0000 5e81 5663 998d 5212 cd0d
/FIRST_TEST/dut/ep21wire b00e 0000 93a5 2c6c 149a d677 b00e
Now 385685 ps
Cursor 1 0 ps
ps 2000000 ps 4000000 ps 6000000 ps 8000000 ps
0 ps

Xilinx ISE iSim

1. Open an ISE Design Suite Command Prompt
2. CD to your WORK DIRECTORY:
   `cd YOUR_WORK_DIRECTORY`
3. Execute the simulation script:
   `sim_isim.bat`



Analyzing the Results

The important stimulus from sim_tf.v comes from the statements within the "initial" block and the tasks called from within that block. In sim_tf.vhd, the important stimulus comes from the statements within the main process after "begin" and the procedures outside the okHostCalls section.

The simulated hardware includes a register that can be used either as a standard counter or as a 32-bit Linear Feedback Shift Register (LFSR). The test fixture uses a TriggerIn endpoint to select the mode and WireIns to seed the register with an initial value. It then reads those values using a WireOut endpoint. Note that the values read are not sequential in this portion of the simulation.

When the test fixture sets the hardware register to "piped" mode, it can read sequential values from the LFSR using a PipeOut endpoint. This is because in "piped" mode the register updates only when the pipe is being read, thus avoiding any potential timing issues that arise when the pipe is interrupted by other processes.

In the USB3 version of the test fixture, the okRegisterBridge endpoint is used to interface with block RAM on the simulated FPGA. The values that are read from the block RAM should be the same as the values that are written to the block RAM. Note that in the VHDL simulation, the process will repeat as long as the simulation is actively running. In Verilog, no further stimulus is applied to the signals once the statements in the initial block have been executed.

Simulation Accuracy

Many of the simulated calls to the FrontPanel host occur more quickly than the equivalent calls that are applied to a physical FPGA. The bandwidth constraints on USB and other operating system issues will cause them to happen much slower. In the interest of simulation speed, however, we have accelerated the response time of some of the host simulation actions. The user may, at his or her discretion, place additional delays within the simulation in order to better model the speed of the real host interface. In most cases, this will not be necessary.

Appendix A: A Simple Example

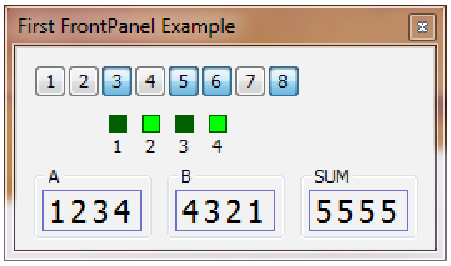
This basic example quickly introduces the basic concepts of the Wire In and Wire Out endpoints by linking real and virtual pushbuttons to real and virtual LEDs. The XML and HDL descriptions are short and concise, making this example a great place to start with FrontPanel.

This sample is available for all FrontPanel devices.

The **First** FrontPanel sample contains the following files:

File	Description
First.xfp	FrontPanel profile (text-readable XML).
first.bit	Xilinx configuration file produced from ISE.
Verilog/First.v	Verilog description of the project's toplevel.
Verilog/First.ucf	Xilinx constraints file containing pin location constraints.

When the profile is loaded into FrontPanel, it creates a user interface that looks like this:



Toplevel Description

The file First.v contains the Verilog description of the project, including all pins which are physically connected to the FPGA. It's entire contents are listed below: (Note that, while the USB version is shown here, the PCIe version is strikingly similar.)

```
module toplevel(
   input  wire [7:0]  hi_in,
   input  wire [1:0]  hi_out,
   inout  wire [15:0] hi_inout,

   output wire [7:0]  led,
   input  wire [3:0]  button
   );

// Target interface bus:
wire        ti_clk;
wire [30:0] ok1;
wire [16:0] ok2;

// Endpoint connections:
wire [15:0] ep00wire;
wire [15:0] ep01wire;
wire [15:0] ep02wire;
wire [15:0] ep20wire;
wire [15:0] ep21wire;

assign led      = ~ep00wire;
assign ep20wire = {12'b0000, ~button};
assign ep21wire = ep01wire + ep02wire;

// Instantiate the okHost and connect endpoints.
wire [17*2-1:0]  ok2x;
okHost okHI(.hi_in(hi_in), .hi_out(hi_out), .hi_inout(hi_inout),
            .ti_clk(ti_clk), .ok1(ok1), .ok2(ok2) );

okWireIn ep00 (.ok1(ok1), .ep_addr(8'h00), .ep_dataout(ep00wire));
okWireIn ep01 (.ok1(ok1), .ep_addr(8'h01), .ep_dataout(ep01wire));
okWireIn ep02 (.ok1(ok1), .ep_addr(8'h02), .ep_dataout(ep02wire));

okWireOut ep20 (.ok1(ok1), .ok2(ok2x[ 0*17 +: 17]),
                .ep_addr(8'h20), .ep_datain(ep20wire));
okWireOut ep21 (.ok1(ok1), .ok2(ok2x[ 1*17 +: 17]),
                .ep_addr(8'h21), .ep_datain(ep21wire));

endmodule
```

Listed inside the module definition are several wires. Most of these are for the FrontPanel host interface. The two other busses, LED[7:0] and BUTTON[3:0] connect to the LEDs and pushbuttons on the XEM3001. Their specific pin locations are constrained in **First.ucf**.

Target Logic

The logic description for this example is very simple and only consists of three lines of HDL connecting the Wire In endpoint to the physical LEDs and the Wire Out endpoint to the physical pushbuttons.

The LEDs are attached to endpoint 0x00 and the pushbuttons are attached to endpoint 0x20. An adder is inferred on two of the Wire Ins (0x01 and 0x02) with the result sent to a Wire Out (0x21).

```
assign led      = ~ep00wire;
assign ep20wire = {12'b0000, ~button};
assign ep21wire = ep01wire + ep02wire;
```

FrontPanel Interface Modules

This design contains three FrontPanel interface modules: okHostInterface, okWireIn, and okWireOut. Their instantiation is pretty straightforward. We have chosen to call the endpoint wires ep00wire and ep20wire for clarity.

FrontPanel XML Description

The user's interface shown at the beginning of this example is described in XML and shown below. Only one instance of the okToggleButton and one instance of the okLED are shown for brevity. The others instances are similar with the exception of their position tag and endpoint bit.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--
First FrontPanel Example
Copyright (c) 2004, Opal Kelly Incorporated
-->

<resource version="2.3.0.1">
<object class="okPanel" name="panel1">
    <title>First FrontPanel Example</title>
    <size>180,70</size>

    <object class="okToggleButton">
        <label>1</label>
        <position>10,10</position>
        <size>20,20</size>
        <endpoint>0x00</endpoint>
        <bit>0</bit>
    </object>

    ... other okToggleButton objects removed ...

    <!-- LEDs -->
    <object class="okLED">
        <position>48,40</position>
        <size>25,25</size>
        <label>1</label>
        <style>SQUARE</style>
        <color>#00ff00</color>
        <endpoint>0x20</endpoint>
        <bit>0</bit>
    </object>

    ... other okLED objects removed ...

</object>
</resource>
```

Each FrontPanel XML description must contain the <?xml> tag shown at the top as well as the <resource ...> and </resource> tags as they are required by the FrontPanel XML parser.

okPanel

The first object specified is the okPanel object which has a "name" property with value "panel1". FrontPanel looks for these properties when loading a profile. They must be sequenced panel1, panel2, and so on. The okPanel object also has two child nodes serving as parameters for the okPanel as listed in the table below:

Node Name	Description
title	This is the title of the dialog window created when you view this panel.
size	The size of the dialog, in pixels: Width,Height.

The okPanel object also has two child nodes which are FrontPanel components, okToggleButton and okLED. Because they are children of the okPanel object, they will appear on this particular panel.

okToggleButton

The toggle button is described with child nodes as indicated in the table below.

Node Name	Description
label	This is a label that will be placed inside the toggle button.
position	The position of the top-left corner of the component, in pixels: X,Y.
size	The size of the component, in pixels: Width,Height.
endpoint	The endpoint address (expressed in hexadecimal) for this toggle button's Wire In endpoint.
bit	The specific bit on the endpoint address that this toggle button controls.

okLED

The LED is described with child nodes as indicated in the table below.

Node Name	Description
label	This is a label that will be placed below the LED.
position	The position of the top-left corner of the component, in pixels: X,Y.
size	The size of the component, in pixels, specified as Width,Height. This size includes the LED and its label.
style	LED style: SQUARE or ROUND
color	The 24-bit color of the LED as #RRGGBB.
endpoint	The endpoint address (expressed in hexadecimal) for this LED's Wire Out endpoint.
bit	The specific bit on the endpoint address that this LED monitors.

okDigitEntry and okDigitDisplay

These two components are described in more detail in the Component XML section of this User's Manual. They provide a convenient way to enter and display multi-bit integers. They support multiple radixes, as well.

Other Samples

The standard FrontPanel installation includes other samples including samples which illustrate use of the C++ and other programmer's interfaces. A summary of these samples is shown below. They are placed in the installation directory in the **Samples** folder.

Sample	FrontPanel	C++	C#	Python	Java	Description
First	✓					A very simple FrontPanel-only project to get started quickly.
Counters	✓	✓		✓		Displays two independent counters with controls for each. Since it is implemented in FrontPanel, C++, and Python, it is a good start for those wanting to learn the APIs.
Controls	✓					This sample is a showcase of the FrontPanel components available.
PipeTest		✓				Connects to PipeIn and PipeOut modules on the FPGA to test transfer rates. Block sizes can be set by the user.
DES	✓	✓	✓	✓	✓	A command-line sample based on the OpenCores.org triple-DES encryption and decryption core.
RAMTester		✓				Command-line sample to read/write FPGA-attached memory.
Flashloader		✓				Command-line sample used to program on-board Flash for FPGA boot configuration.

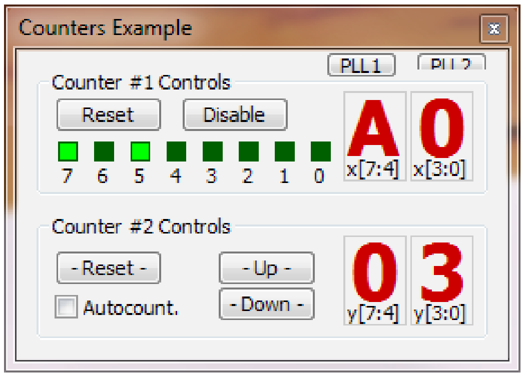
Appendix B: The Counters Sample

This sample is a bit more complicated than the simple example and showcases a few more features of FrontPanel and the XEM3001.

This sample is designed to work with the XEM3001.

The Counters sample is a bit more complicated than the previous example. It includes a few more FrontPanel components and also adds a few Trigger endpoints. More importantly, though, it adds more hardware in the form of HDL so you can see how FrontPanel integrates with HDL in a slightly more complicated setup.

The FrontPanel virtual interface for this sample is shown below:



Hardware Description

The hardware for the Counters sample has two counters, the okHostInterface, a single Wire In endpoint, three Wire Out endpoints, and a Trigger In endpoint. The hardware also routes to the LEDs on the XEM3001.

Counter #1

The first counter is an 8-bit up counter with enable, synchronous reset, and disable. The enable signal is generated by a separate 24-bit counter to make the count progression slower. The Verilog HDL for this counter and its clock divider counter is shown here:

```
always @(posedge clk1) begin
   div1 <= div1 - 1;
   if (div1 == 24'h000000) begin
      div1 <= 24'h400000;
      clk1div <= 1'b1;
   end else begin
      clk1div <= 1'b0;
   end

   if (clk1div == 1'b1) begin
      if (reset1 == 1'b1)
         count1 <= 8'h00;
      else if (disable1 == 1'b0)
         count1 <= count1 + 1;
   end
end
```

From the description, we gather that when RESET1 is asserted, the counter will hold the value 0x00. When DISABLE1 is asserted, the counter holds its current value. Otherwise, the counter will increment each time the clock divider counter expires.

Note that this counter operates on CLK1 which is mapped to LCLK1 on the PLL.

Counter #2

The second counter operates on CLK2 which is mapped to LCLK2 on the PLL. Using the PLL Configuration Dialog, we will be able to observe the effects of changing the PLL frequencies on the two counters.

The Verilog HDL for this counter and its own divider is listed below. This counter will count up when UP2 is asserted, count down when DOWN2 is asserted, and automatically count up when AUTOCOUNT2 is asserted. Note that UP2 and DOWN2 must be asserted for exactly one CLK2 cycle for the counter to count only one. This is why we have the Trigger endpoints.

```
always @(posedge clk2) begin
   div2 <= div2 - 1;
   if (div2 == 24'h000000) begin
      div2 <= 24'h100000;
      clk2div <= 1'b1;
   end else begin
      clk2div <= 1'b0;
   end

   if (reset2 == 1'b1)
      count2 <= 8'h00;
   else if (up2 == 1'b1)
      count2 <= count2 + 1;
   else if (down2 == 1'b1)
      count2 <= count2 - 1;
   else if ((autocount2 == 1'b1) && (clk2div == 1'b1))
      count2 <= count2 + 1;
end
```

Endpoints

This sample uses several endpoints to provide controllable inputs to the hardware and observable outputs to FrontPanel. To reduce the number of endpoints, we have chosen to share them among the counters.

Wire In (0x00)

The only Wire In endpoint is used to carry the RESET1, DISABLE1, and AUTOCOUNT2 signals. These are wires because we want them to have a static state rather than one-shot signals.

Signal	Bit(s)	Description
RESET1	0	When asserted, Counter #1 holds the value 0x00 and does not count.
DISABLE1	1	When asserted, Counter #2 holds its value and does not count.
AUTOCOUNT2	2	Configures counter #2 to autocount.
Unused	15:3	

Trigger In (0x40)

The only Trigger In endpoint is used for the Counter #2 inputs. These are triggers because we want single events (one-shots) to occur, such as a count-up event.

Note that RESET2 behaves the same as RESET1 but we want to have RESET2 behave as a one-shot event so that the user cannot hold RESET2 asserted. Therefore, we attach this one to a Trigger.

Signal	Bit(s)	Description
RESET2	0	When asserted, Counter #2 resets to 0x00 and does not count.
UP2	1	When asserted, Counter #2 counts up.
DOWN2	2	When asserted. Counter #2 counts down.
Unused	15:3	

Wire Out (0x20, 0x21, and 0x22)

These wires provide observables for FrontPanel. They are connected as follows:

Endpoint	Signal	Description
Wire Out 0x20	COUNT1[7:0]	Counter #1 value.
Wire Out 0x21	COUNT2[7:0]	Counter #2 value.
Wire Out 0x22	BUTTON[3:0]	The lower four bits of this wire bundle contain the status of the on-board pushbuttons. If a button is pressed, the corresponding wire will be asserted.

FrontPanel Components

The user interface for the Counters sample includes two panels. The first panel contains five buttons, four hex displays, eight LEDs, and a check box. There are also two cosmetic components called okStaticBox which are used to group the components visually. The second panel simply contains four LEDs used to display the state of the pushbuttons.

Panel 1: Counters Example

The active FrontPanel components are listed below with their corresponding endpoints:

Component	Label	Endpoint	Bit
okPushbutton	Reset	0x00	0
okPushbutton	Disable	0x00	1
okTriggerButton	- Reset -	0x40	0
okTriggerButton	- Up -	0x40	1
okTriggerButton	- Down -	0x40	2
okToggleCheck	Autocount.	0x00	2
okHex	x[7:4]	0x20	4
okHex	x[3:0]	0x20	0
okHex	y[7:4]	0x21	4
okHex	y[3:0]	0x21	0
okLED	7	0x20	7
okLED	6...1	0x20	6...1
okLED	0	0x20	0

Note that the okLED and two of the okHex components share endpoint 0x20. FrontPanel allows this and will update both components when Wire Out endpoints change. It is also possible to map two components to input endpoints.

Panel 2: Pushbuttons

The second panel is not automatically opened when the Conters XFP file is loaded. You can open it by pressing the number `2' on your keyboard or navigating to

View → Pushbuttons

at the top of the FrontPanel window. This displays a small window with the following components:

Component	Label	Endpoint	Bit/Mask
okLED	3	0x22	3
okLED	2	0x22	2
okLED	1	0x22	1
okLED	0	0x22	0