BeastLink API Specification

The BeastLink API is the host component to communicate between a host PC and AXI-4 peripherals inside a Xilinx 7 Series FPGA over USB using an Cypress FX3. The API is supported on different operating systems. BeastLink consists of a software library that is used in the host application, drivers components, Cypress FX3 firmware and an IP Core that is embedded in the FPGA design on the target board.

☑ Free Edition

☑ Pro Edition

BeastLink Features

BeastLink is a cross-platform framework to easily transfer data between a PC and AXI-4 peripherals without need to understand any component in between. This means, no knowledge of USB access on the host side, including drivers, USB transfers, Cypress FX3 firmware or the interface between Cypress FX3 and the Xilinx 7 Series FPGA is required. Just connect the Xilinx 7 Series FPGA to the Cypress FX3 as documented on your own hardware design, add our IP Core into your Xilinx Vivado design suite project and transfer data between host and AXI-4 peripherals with few lines of source code. Consult the individual documents for specific information.
It provides numerous advantages to system designers and end-users:

- Unique cross-platform access layer to supported hardware
- Transparency of the underlying bus system functionality
- Hide the complexity of system- and bus-specific implementations
- No throughput-reduction of the subjacent bus system
- Support for different programming languages (C, C++, Python and all .NET and JVM languages)

The most important functionalities of BeastLink are:

- Simple device enumeration and access
- Address based communication with the AXI-4 bus
- Consistent error handling
- Serialized access to components

BeastLink is designed to operate on Microsoft © Windows and Linux.

API design

With only one exception, the API has a nearly identical interface to all supported programming languages. There are only minor differences based on specific language features and to stay compatible with the respective coding guidelines (e.g. case conventions). Devices can easily be accessed in an object orientated manner. Each enumerated hardware device is enumerated by the API as instance of class Device. This class has functionality to handle all supported features.

The only difference is C, which has no object orientated features. The C interface directly uses the core layer without the need for a different abstraction. All other languages have a thin layer between application and core API.

Thread safety

The API is designed to be thread safe. Global functionality and device access is exclusively locked. The only exceptions are functions that are device related. They have a shared lock, so different threads can independently communicate with different devices.

Using BeastLink API with Microsoft © Windows

BeastLink is tested and supported on Windows 7 x64 and Windows 10 x64. Windows Vista, 8 and 8.1 as well as all 32 bit versions should be compatible as well, but are not officially supported.

The core API component of BeastLink is available as 32 and 64 bit binary. Usage depends on the used compiler or framework, not the OS architecture. All programming languages with exception to C/C++ decide at runtime, which library is used (which depends on the bitness of their used virtual machine). Libraries are named **beastlink-1.0-x86.dll** and **beastlink-1.0-x86_64.dll**. This file must be available at runtime in one of Windows shared library loading locations.

Driver and service

On Windows systems, Microsoft © WinUSB is used as driver for USB devices. This guarantees compatibility on all supported Windows versions. The driver installation is a configurable component of the BeastLink framework.

The service is used to download the Cypress FX3 firmware on device connection. This is done by hooking onto a device connection event. If a known USB device is connected to the PC (based on the service configuration file), the respective firmware is downloaded and the device is rebooted using a masked PID.

Using BeastLink with Linux

BeastLink is tested and officially supported on the latest official LTS version of Ubuntu Desktop, x86_64. Other Linux distributions and version are expected to be compatible as well.

The core API component of BeastLink is a shared library called **libbeastlink-1.0.so**. This file must be in one of the paths that is search for shared libraries. One common possibility is to put the file into the startup library of the application and extend LD_LIBRARY_PATH to this directory.

Driver and udev-rule

All communication on Linux systems is done using libusb 1.x. This grants compatibility to nearly all Linux derivatives which offer this interface. No specific driver module is required, all things are handled in user space.

To use the devices on Linux systems, a udev rule must be installed, which is responsible for two important tasks:

- Non-root users get access to the devices, the rule sets the permission for all devices to 666 (rw-rw-rw-).
- Upon connection of a device, a firmware download tool is called, which downloads the firmware to the Cypress FX3 and reboots the device. Without this firmware, the devices are not usable.

The installation of this rule is done using shell script **install-usb.sh** which must be started as root user. This installs the rule as **99-beastlink-permissions.rules** into the user specified udev rule directory. The install script creates a script called **uninstall.sh**, which removes to rules from the system.
If the installation path changes, please update the rule!

BeastLink API

Important files

API Path	Description
C	
c/example.c	Example for C API.
c/lib/*	Files required for linking when using Microsoft Visual C++ (2015 or higher).
c/include/beastlink.h	Header for C API.
C++	
c++/example.cpp	Example for C++ API.
c++/lib/*	Files required for linking when using Microsoft Visual C++ (2015 or higher).
c++/include/beastlink.h	Header for C API.
c++/beastlink++.h	Header file for C++ API.
c++/beastlink++.cpp	C++ API implementation. Must be included in C++ projects the use BeastLink.
JVM	
jvm/beastlink-1.0.jar	The library layer for JVM. Example in Java included.
jvm/beastlink-1.0-sources.jar	Sources for the JVM interface, written in Java.
NET	
net/example/*	Example for .NET API in C#.
net/api/*	C# sources for the NET interface, written in C#.
Python	
python/example.py	The python example.
python/beastlink.py	Library layer for Python. Compatible with Python 2.x and 3.x.

BeastLink language support

BeastLink supports different programming languages. The core layer is a shared library, which is a .dll file on Windows and a .so file on Linux. It offers all functionality as functions using standard types. New languages can be added using a thin language specific layer. The main design goal was to offer a similar interface for each programming language by preserving the idioms and conventions of the language in question. Except the C interface, all languages access the functionality in an object-oriented manner. The OS-specific runtime files must be accessible at run-time for all used programming languages. It is recommended to take a look at the example which is available for every supported language. They are well documented and show most features of the API.

C

Language specific

The C interface is the only one that accesses the core layer directly. All functions are prefixed with Bl, constants are prefixed using BL. Most functions return an error code which should be checked in any case. If an error has been detected, the error reason can be retrieved from **BlGetLastErrorText()**. As this interface does not offer object orientated design, devices are referenced using a handle. The handle is retrieved when opening a device and must be used subsequently to specify the device in question until **BlClose()** is called.

Build

BeastLink can be used by including **beastlink.h** into the project. When using Visual Studio on Windows, **beastlink-1.0-x86.lib** must be added as link library for 32 bit programs and **beastlink-1.0-x86_64.lib** for 64 bit programs. For Linux, the shared library **libbeastlink-1.0.so** must be used in the linking stage.

Object-orientated languages

C++, Python and all .NET and JVM languages offer the possibility to use the BeastLink interface using objects. Besides minor differences, all have the same interface. The following table shows the types relevant for the API user:

BeastLink API types	
DeviceInfo	Offers various information about the device it is mapped to.
EnumeratedDevice	This type is returned for every device found during enumeration. They are only valid until a new enumeration is started.
Device	Returned from EnumeratedDevice.open(). The object to interact with the hardware.
LibraryInterface	As the .NET and JVM framework do not directly support any global methods, this type contains global functionality, which are functions in C++ and Python.

C++

Language specific

Language compatibility: C++14

Official supported compilers: Visual C++ 2015 (Windows), G++ >= 5.4.0 (Linux)

The C++ interface is compatible to C++14 and uses elements of the standard template library (STL) wherever suitable. All functionality is encapsulated in namespace **beastlink** to not interfere with other API's. Error handling is done using exceptions. Whenever the core layer reports an error, the C++ layer reads the textual reason and throws a **std::exception**.

Build

BeastLink can be used by adding **beastlink++.cpp** into the project. The interface is accessible by including **beastlink++.h** wherever required. When using Visual Studio on Windows,

beastlink-1.0-x86.lib must be added as link library for 32 bit programs and **beastlink-1.0-x86_64.lib** for 64 bit programs. For Linux, the shared library **libbeastlink-1.0.so** must be used in the linking stage.

JVM

Language specific

Language compatibility: Java SE 8 or higher

The JVM interface is written in Java and included as pre-built jar. The JNA library is used to interact with the core API at runtime.
All errors are reported using **java.io.IOException**.
The JVM port has a special method to load an FPGA design directly from the .jar to the device: **Device.programFpgaFromResource()**.

Build

Just add **beastlink-1.0.jar** to your project. In addition JNA library must be available as well.

NET framework

Language specific

Language compatibility: .NET framework 3.5 and higher

The interface is written in C# but usable with all .NET compatible languages. The wrapper is successfully tested with mono, which allows cross-platform development.
Errors are reported using **System.IO.IOException**.
Wherever suitable, properties are used instead of Get/Set methods. In comparison to the other languages, Pascal case is used for method names to preserve the CLR guideline.
The API can be built with the free Community versions of Visual Studio and Mono.

Build

Add **beastlinknet-1.0.jar** to the application that is using BeastLink.

Python

Language specific

Language compatibility: Both 2.x and 3.x branches.

Errors are reported by raising an **Exception**.

General API overview

The following table lists all API functions and their relation across all supported programming languages.

The functions are grouped into 3 categories:

- Global, device independent functions like API initialization, cleanup, error handling and device enumeration.
- Device related functions like device preparation and data transfer.
- Device information functions to access device data like serial number and user ID.

Global, device independent functions

These functions are either global (C++, Python) or reside statically in class **LibraryInterface** (JVM, .NET).

C	C++, JVM, Python	.NET
BlGetLastErrorText		
BlGetVersion	getUdkVersion	UdkVersion
BlInit	init	Init
BlEnumerate	enumerate	Enumerate
BlCleanup	cleanup	Cleanup
BlSetLogLevel	setLogLevel	SetLogLevel

Device related functions

These functions reside in class **Device**, except **open()** which is member of class **EnumeratedDevice**.

C	C++, JVM, Python	.NET
BlOpen	open	Open
BlClose	close	Close
BlReadRegister	readRegister	ReadRegister
BlWriteRegister	writeRegister	WriteRegister
BlReadBlock	readBlock	ReadBlock

C	C++, JVM, Python	.NET
BlWriteBlock	writeBlock	WriteBlock
BlResetFpga	resetFpga	ResetFpga
BlProgramFpgaFromBin	programFpgaFromBin	ProgramFpgaFromBin
BlProgramFpgaFomMemory	programFpgaFromMemory	ProgramFpgaFromMemory
BlSetTimeout	setTimeout	SetTimeout

Device information functions

These functions reside in class **DeviceInfo**.

C	C++, JVM, Python	.NET
BlSetUserId	setUserId	UserId
BlGetUserId	getUserId	UserId
BlGetDerivateInfo	getDerivateInfo	DerivateInfo
BlGetDerivateId	getDerivateId	DerivateId
BlGetSerialNumber	getSerialNumber	SerialNumber
BlGetFirmwareVersion	getFirmwareVersion	FirmwareVersion

Lifecycle of an application using BeastLink

If not explicitly stated, the method names in this chapter are used from C++, JVM and Python.

Initialization

The first task to do is the initialization of the API. This calls startup code of the underlying frameworks. Without doing this, functionality of the API is not granted. So calling **init()** should be the first thing using BeastLink.

Enumeration and Open

Devices can now be enumerated. With the exception of C, this is simply done calling **enumerate()**, which returns a list of devices found in the system. Devices returned are only those who are not already opened (system wide). This method expects vendor and product ID (VID, PID) of the USB devices to search for. The returned list contains elements of type **EnumeratedDevice*. This type can be understand as a possible "device candidate". Calling the **EnumeratedDevice.open()** method tries to connect to the device and returns an instance of type Device in case of success. The invocation of **EnumeratedDevice.open()** can fail, if a different application has opened the device in the time between enumeration and opening it (or the device has been unplugged between these calls).
This process can be done for multiple devices. If a new enumeration is done using **enumerate()**, all previous instances of **EnumerateDevice** are invalid. Instances of type Device are not affected!

[C specific] In C, an enumeration is done calling **BlEnumerate()**, which returns the count of "device candidates". **BlOpen()** returns the handle for subsequent usage. This enumeration is shown in the example application (example.c). If **BlEnumerate()** is called again, the count and enumeration ID's from a previous call are invalid. Handles returned by **BlOpen()** are not affected!
At this point, all previously unavailable information (user ID, serial number size, derivate info and ID) can be accessed (through **Device.getDeviceInfo()**).

FPGA Configuration

Device communication can be done using the **Device** instance [C language: handle]. To configure an FPGA with a configuration bitstream from the host, one of the possible **Device.program*()** methods should be used (Device.programFpgaFromBin() or Device.programFpgaFromMemory(). The file format for FPGA-Designs is .bin (not .bit !). In Xilinx Vivado design suite, the .bin file can be generated by selecting the “-bin_file*” option under “Flow Navigator/PROJECT MANAGER/Settings/Bitstream”. If Vivado TCL Console is being used to generate a bitstream, then the option “-bin_file” must be added to the “write_bitsream” command.

If an FPGA design is already loaded (using JTAG or loaded from flash), a call to **Device.resetFpga()** must be done to synchronize communication with the host. Without an active FPGA design, data transfer will time out.

Read and Write

Data transfer with the AXI-4 peripherals is done using **Device.readBlock()** and **Device.writeBlock()**. The addresses, sizes and flags must match the implementation inside the FPGA. **Device.readRegister()** and **Device.writeRegister()** are convenience methods that transfer 4 bytes of data using the same mechanics and offer the input and output value as 32 bit unsigned integer. If a transfer takes longer than the time specified using **Device.setTimeout()**, it is recognized as failed. Under some circumstances (e.g. transfer to slow peripherals), this value must be adjusted. The default value is 1000 milliseconds. If the connection to the device gets lost (e.g. unplug), the next call to one of these communication methods will fail, there's no other way to get informed about unplugging.

Close Device

If the device communication isn't needed anymore, calling **Device.close()** will close the connection and make the device available for new enumerations again.

A call to **cleanup()** will close all devices and completely cleanup all internal structures. It is possible to start with **init()** at this point again.

Revision history

Version	Date	Comment	Author	Approved
1.0	Feb, 26 2018	Initial release	th	mr

Copyright Notice

Address

CESYS Gesellschaft für angewandte Mikroelektronik mbH
Gustav-Hertz-Str. 4
D - 91074 Herzogenaurach
Germany