UDK3 API Specification

The Cesys Unified Development Kit Release 3 (UDK3) is a concept for connecting CESYS FPGA boards with different host-interfaces to hosts running different operating systems. The modular structure of the UDK3 makes it easy to switch the operating systems or the host interfaces you wish to use in your project and to support any combination. UDK3 consists of a software library that is used in the host application and a IP that is embedded in the FPGA design of the target board.

UDK3 Features

The UDK3 is a cross-platform concept that has modularized interfaces to various Cesys devices[1].
It's design enables communication using various bus systems. Its current version implements the USB interface[2].

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

The most important functionalities of the UDK3 core are:

- simple device enumeration and access
- access to all information about a specific device
- address based communication even with serial interfaces like USB or Ethernet.
- consistent error handling

UDK3 is designed to operate on Microsoft™ Windows, Linux and Apple™ Mac OS X.

Thread safety

The API is generally thread safe. Most functionality 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. If more than one thread tries to access a single device, the behavior is undefined.

1 There is also a license available to adopt customer boards and to use and distribute the UDK3 binaries with your own hardware. Ask for UDK3 in-house source-code license.
2 More interfaces will be added to the UDK3 as new Cesys boards become available. The existing Cesys boards with PCI and PCIe interface will stick with UDK2.

Obtaining and installing UDK3

UDK3 can be downloaded from www.cesys.com. It is distributed as single archive file, but available as .zip and .tar.bz2. There is no installation required. Please read the instructions in the README provided for each programming language for detailed usage instructions.

UDK3 Software structure

The UDK3 structure splits into different abstraction layers:

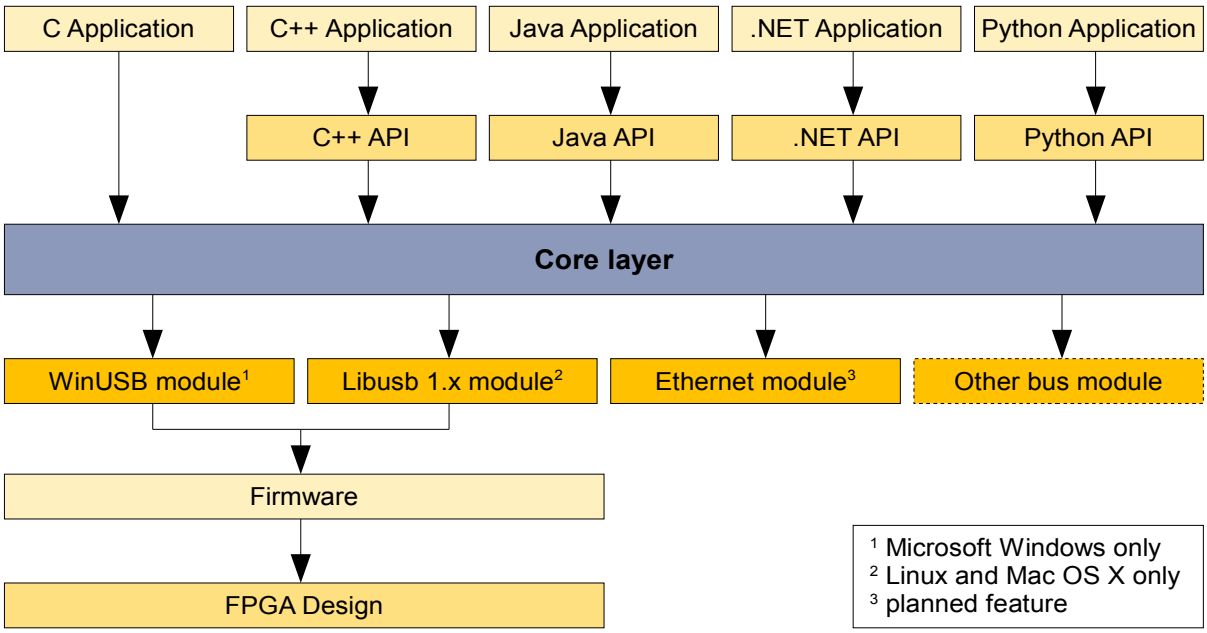

Figure 1: UDK3 Layers

Applications can access the core layer directly using the C – API or through language specific layers (which offer the API in an object orientated style).
All language specific layers are provided as source code.

The core layer of the UDK3 is a communication-bus neutral component. It loads all compatible modules during startup and acts as gateway to all of them.
For example, the enumeration of devices returns a list that contains the devices that has been found in all modules. All subsequent communication with an opened device is routed by the core layer to the specific module.

Using UDK3 with Microsoft™ Windows

Windows support ranges from Windows Vista up to Windows 8.1.

Runtime libraries

The files in the table below must be available at runtime, independent to the used programming language. udk3mod*.dll files must either be in the startup path of the application or in a path that must be explicitly communicated to the API.

Library file	
udk3-1.1.dll	The core layer.
udk3mod-1.1-winusb.dll	WinUSB module.

USB

On Windows systems, Microsoft™ WinUSB is used as driver for USB devices. This guarantees compatibility on all supported Windows versions.

To use USB devices on Windows, a driver must be installed for the device. In addition to the driver, a background service downloads the firmware to newly connected devices, so they can be used. The service can be found in the service panel as **Cesys UDK3 USB Device Service**. If this service is stopped, devices remain **unconfigured** and can't be used.

Both driver and service are part of the separate UDK3 USB driver installation package which is intended to be installed on end user PC's.

Using UDK3 with Linux

We tested UDK3 with different Linux distributions on x86_64 processor architecture. The primary testing platform is the latest LTS version from Ubuntu (14.04). An ARMv6 module, tested on Raspberry Pi, is also available.

Runtime libraries

The files in the table below must be available at runtime in the library path, independent to the used programming language. udk3mod*.so files must either be in the startup path of the application or in a path that must be explicitly communicated to the API. Best practice is to put all these files in the startup path of the application and extend environment variable LD_LIBRARY_PATH to this folder.

Library file	
libudk3-1.1.so	The core layer.
libudk3mod-1.1-libusb.so	libusb 1.x module.

USB

All communication on Linux systems is done using libusb 1.x. This grants compatibility to nearly all Linux derivatives which offer this interface.

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 Cesys UDK3 devices to 666 (rw-rw-rw-).
- Upon connection of a Cesys UDK3 device, a firmware download tool is called, which downloads the firmware to 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-udk-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 !

Using UDK3 with Apple™ Mac OS X

OS X support is planned for future releases.

UDK3 API

Important files

API Path	Description
C / C++	
c_c++/examples/*	Examples for C and C++ API.
c_c++/lib/*	Files required for linking when using Microsoft Visual C++.
c_c++/udk3api.h	Header for for the C API
c_c++/udk3api++.h	Header file for the C++ API.
c_c++/udk3api++.cpp	C++ API implementation. Must be included in C++ projects the use the UDK3.
c_c++/*.cmd	Batch files to create Visual Studio C++ projects or MinGW Makefiles.
c_c++/*.sh	Shell scripts to create Makefiles on POSIX platforms (Linux / Mac OS X).
Java	
java/udk3api-1.0.jar	The library layer for Java. Example included.
java/udk3api-1.0-sources.jar	Sources for the Java interface.
.NET	
net/example/example.cs	Example for .NET API in C#.
net/udk3api/*	C# sources for the .NET interface. Compile them for your .NET framework.
net/*.cmd	Batch files to build Visual Studio Solution on Windows (compatible with Monodevelop).
net/*.sh	Shell scripts to create a Monodevelop solution on POSIX platforms (Linux / Mac OS X).
Python	
python/example.py	The python example.
python/udk3api.py	Library layer for Python. Compatible with Python 2.x and 3.x.

UDK3 Language support

UDK3 supports five different programming languages. As the core layer is a shared library (Windows: .DLL, Linux: .SO and Mac OS X: .DYLIB), which 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 object orientated manner.

Independent to the used language: At runtime, the OS specific runtime files must be accessible.

Best practice is 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 directly accesses the core layer.

All functions are prefixed with **Ce**, constants are prefixed using **CE**.

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 **CeGetLastErrorText()**.

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 **CeClose()** is called.

Build

UDK3 can be used by including **udk3api.h** into the project.

When using Visual Studio, **udk3-1.1.lib** must be added as link library. For all other compilers, the system specific shared library must be used in the

linking stage (Runtime libraries as specified in the Operating system support section).

Object orientated languages

C++, Java, .NET and Python offer the UDK3 interface using objects. Besides minor differences, all have the same interface. The following table shows the types relevant for the API user:

UDK3 types	
BusType	An enumeration for all supported buses.
DeviceType	An enumeration of known devices or device classes.
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 .NET and Java do not support any global methods, this type contains global functionality, which are functions in C++ and Python.

C++

Language specific

The C++ interface is compatible to C++03 and uses elements of the standard template library (STL) wherever suitable.

All functionality is encapsulated in namespace **udk3api** to not interfere with other API's.

Error handling is done using exceptions. Whenever the core layer reports and error, the C++ layer reads the textual reason and throws a **std::exception**.

Build

UDK3 can be used by adding **udk3api++.cpp** into the project. The interface is accessible by including **udk3api++.h** wherever required.

When using Visual Studio, **udk3-1.1.lib** must be added as link library. For all other compilers, the system specific shared library must be used in the linking stage (Runtime libraries as specified in the Operating system support section).

Java

Language specific

The Java interface is prebuilt for Java 1.7.

There's an external dependency to **jna**. The latest version can be found here: https://github.com/twall/jna.

All errors are reported using **java.io.IOException**.

Java has a special method to load an FPGA design directly from the .jar to the device: Device.programFpgaFromResource().

Build

Just add **udk3api-1.1.jar** and **jna** to your project.

.NET / CLR

Language specific

The interface is written in C# but usable with all .NET compatible languages. The interface is successfully tested with **mono**, so cross platform development is possible.

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 interface is compatible to .NET framework 4.0 and higher. Older versions may be compatible but this is not verified.
The API can be built with the free Express versions of Visual Studio and Mono.

Build

Add **udk3apinet-1.1.dll** to the application that requires UDK3.

Python

Language specific

The interface is compatible to both 2.x and 3.x branches. As Python has no native support for enumerators, bus- and device types are defined globally.

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, deinitialization, 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.

C	C++	Java	.NET	Python
Global, device independent functions				
CeGetLastErrorText	-[3]	-[3]	-[3]	-[3]
CeGetUdk3VersionString	getUdkVersion	LibraryInterface.getUdkVersion	LibraryInterface.UdkVersion	getUdkVersion
CeInit	init	LibraryInterface.init	LibraryInterface.Init	init
CeInitEx	initEx	LibraryInterface.initEx	LibraryInterface.InitEx	initEx
CeEnumerate	enumerate	LibraryInterface.enumerate	LibraryInterface.Enumerate	enumerate
CeEnumerateInfo	-[4]	-[4]	-[4]	-[4]
CeDeInit	deInit	LibraryInterface.deInit	LibraryInterface.DeInit	deInit
CeSetLogLevel	setLogLevel	LibraryInterface.setLogLevel	LibraryInterface.SetLogLevel	setLogLevel
Device related functions				
CeOpen	EnumeratedDevice.open	EnumeratedDevice.open	EnumeratedDevice.Open	EnumeratedDevice.open
CeClose	Device.close	Device.close	Device.Close	Device.close
CeReadRegister	Device.readRegister	Device.readRegister	Device.ReadRegister	Device.readRegister
CeWriteRegister	Device.writeRegister	Device.writeRegister	Device.WriteRegister	Device.writeRegister
CeReadBlock	Device.readBlock	Device.readBlock	Device.ReadBlock	Device.readBlock
CeWriteBlock	Device.writeBlock	Device.writeBlock	Device.WriteBlock	Device.writeBlock
CeWaitForInterrupt	Device.waitForInterrupt	Device.waitForInterrupt	Device.WaitForInterrupt	Device.waitForInterrupt
CeEnableInterrupt	Device.enableInterrupt	Device.enableInterrupt	Device.EnableInterrupt	Device.enableInterrupt
CeResetFpga	Device.resetFpga	Device.resetFpga	Device.ResetFpga	Device.resetFpga
CeProgramFpgaFromBin	Device.programFpgaFromBin	Device.programFpgaFromBin	Device.ProgramFpgaFromBin	Device.programFpgaFromBin
CeProgramFpgaFomMemory	Device.programFpgaFromMemory	Device.programFpgaFromMemory	Device.ProgramFpgaFromMemory	Device.programFpgaFromMemory
CeProgramFpgaFomMemoryZ	Device.programFpgaFromMemoryZ	Device.programFpgaFromMemoryZ	Device.ProgramFpgaFromMemoryZ	Device.programFpgaFromMemoryZ

3 Not accessible as error handling is automatically done by the respective language layer using exceptions.
4 CeEnumerate and CeEnumerateInfo are used inside enumerate().

C	C++	Java	.NET	Python
CeSetTimeOut	Device.setTimeout	Device.setTimeout	Device.SetTimeout	Device.setTimeout
CeEnableBurst	Device.enableBurst	Device.enableBurst	Device.EnableBurst	Device.enableBurst
Device information functions				
CeSetUserId	DeviceInfo.setUserId	DeviceInfo.setUserId	DeviceInfo.UserId	DeviceInfo.setUserId
CeGetUserId	DeviceInfo.getUserId	DeviceInfo.getUserId	DeviceInfo.UserId	DeviceInfo.getUserId
CeGetDerivateInfo	DeviceInfo.getDerivateInfo	DeviceInfo.getDerivateInfo	DeviceInfo.DerivateInfo	DeviceInfo.getDerivateInfo
CeGetDerivateId	DeviceInfo.getDerivateId	DeviceInfo.getDerivateId	DeviceInfo.DerivateId	DeviceInfo.getDerivateId
CeGetMaxTransferSize	DeviceInfo.getMaxTransferSize	DeviceInfo.getMaxTransferSize	DeviceInfo.MaxTransferSize	DeviceInfo.getMaxTransferSize
CeGetSerialNumber	DeviceInfo.getSerialNumber	DeviceInfo.getSerialNumber	DeviceInfo.SerialNumber	DeviceInfo.getSerialNumber
CeGetFirmwareVersion	DeviceInfo.getFirmwareVersion	DeviceInfo.getFirmwareVersion	DeviceInfo.FirmwareVersion	DeviceInfo.getFirmwareVersion

Lifecycle of an application using UDK3

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

Initialization

The first task to do is the initialization of the API. This prepares some internal structures and loads all UDK3 modules.
There are two different calls to accomplish this, **init()** or **initEx()**. While **init()** loads the modules from the current path, the call to **initEx()** allows an explicit specification for a path were modules should be loaded from. This can be useful if there are any requirements to directory structures.

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 a device type or device class to search for and filters the list of detected devices by this parameter.
The returned list contains elements of type **EnumeratedDevice**. This type can be understand as a possible “device candidate”. It's instance contains some device information in it's device information structure (**EnumeratedDevice.getDeviceInfo()**).
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 **CeEnumerate()**, which returns the count of “device candidates”. For every device **CeEnumerateInfo()** must be invoked to gather information about the instance. **CeOpen()** returns the handle for subsequent usage. This enumeration is shown in the example application (**example.c**).
If **CeEnumerate()** is called again, the count and enumeration ID's from a previous call are invalid. Handles returned by **CeOpen()** are not affected !

At this point, all previously unavailable information (user ID, serial number, max transfer size, derivate info and ID) can be accessed (through **Device.getDeviceInfo()**).

FPGA Configuration

Device communication can be done using the **Device** instance [C: handle]. To configure a FPGA with a configuration bitstream by the host, one of the three possible **Device.program*()** methods should be used (**Device.programFpgaFromBin()**, **Device.programFpgaFromMemory()** or **Device.programFpgaFromMemoryZ()**).

The file format for FPGA-Designs is .bin (not .bit !). With ISE, check option **Create binary configuration file** in **Programming File / Process Properties**.

To convert a .bit to .bin using the SDK, choose **Xilinx Tools / Launch Shell**. Change to the directory of the .bit and call **promgen -u 0 [design].bit -p bin -spi -w** , where **[design].bit** is the input file, and **[design].bin** is generated.

When a 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 a active FPGA design, data transfer will time out.

Read and Write

Data transfer with the FPGA design is done using **Device.readBlock()** and **Device.writeBlock()**. The addresses, sizes and flags must match the FPGA implementation.
FPGA designs may be different when using varying bus systems (USB / PCI / Ethernet), but on the host side they are always the same. This was one of the primary design goals for UDK3!
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.

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

Device communication

As not all buses have a unique address based protocol, UDK3 tries to replicate this feature using different ways.

USB

USB has no native support for address based communication the way UDK3 offers it. In this case, a simple protocol sits on top of the communication. This is handled on the host side by the UDK3 internally, API users don't have to care about that.
On device / FPGA side, the implementation is described in application note AN101 UDK3 Transfer Protocol.

Copyright Notice

Disclaimer

CESYS Gesellschaft für angewandte Mikroelektronik mbH
Zeppelinstrasse 6a
D - 91074 Herzogenaurach
Germany

Revision history

v1.0	February, 21 2014	Initial release.
V1.1	May, 13 2014	UDK 1.1 release fixes.

Table of contents