UDK - Python Interface

Copyright information

Copyright © 2010 CESYS GmbH. All Rights Reserved. The information in this document is proprietary to CESYS GmbH. No part of this document may be reproduced in any form or by any means or used to make derivative work (such as translation, transformation or adaptation) without written permission from CESYS GmbH.

CESYS GmbH provides this documentation without warranty, term or condition of any kind, either express or implied, including, but not limited to, express and implied warranties of merchantability, fitness for a particular purpose, and non-infringement. While the information contained herein is believed to be accurate, such information is preliminary, and no representations or warranties of accuracy or completeness are made. In no event will CESYS GmbH be liable for damages arising directly or indirectly from any use of or reliance upon the information contained in this document. CESYS GmbH will make improvements or changes in the product(s) and/or program(s) described in this documentation at any time.

CESYS GmbH retains the right to make changes to this product at any time, without notice. Products may have minor variations to this publication, known as errata. CESYS GmbH assumes no liability whatsoever, including infringement of any patent or copyright, for sale and use of CESYS GmbH products.

CESYS GmbH and the CESYS logo are registered trademarks.

All product names are trademarks, registered trademarks, or service marks of their respective owner.

⇒ Please check www.cesys.com to get the latest version of this document.

CESYS Gesellschaft für angewandte Mikroelektronik mbH

Zeppelinstrasse 6a

D – 91074 Herzogenaurach

Germany

UDK Python Interface

Introduction

The UDK Python Interface consists of two parts. On the one hand a dynamic link library, (Windows: “udk_python_if.pyd”, Linux: “udk_python_if.so”), which is a wrapper for the UDK API functions. It calls the methods internally and exports them in Python way. And on the other hand a Python class, “udk_python_if_class.py”, which simplifies the access to the functions exported from the dynamic link library. It approximates the usage of the python functions to the usage of the original UDK API functions.

Windows

Requirements

- Python 3.1.2 → www.python.org

Note: The UDK Python Interface will not work with Python 2 as Python 3 is backwards-incompatible.

The working directory for own applications must contain

• UDK C++ API (built version)	→ udkapi_vc[ver]_[arch].dll
• Python dynamic link library	→ udk_python_if.pyd
• Python class definition	→ udk_py_if_class.pyc

To build the interface the following is also required:

- CMake 2.6 or higher → www.cmake.org
- Microsoft Visual Studio 2005 or 2008; 2010 is experimental

Build the UDK Python Interface dynamic library

Prerequisites

There are some options that need to be fixed in *msvc.cmake* inside the UDK Python Interface root directory:

- **PYTHON_ROOT** Path to the Python root directory
- **USE_STATIC_RTL** If *0*, all projects are build against the dynamic runtime libraries. This requires the installation of the appropriate Visual Studio redistributable pack on every machine the UDK Python Interface is used on.
- **UDK_ROOT** Path to the UDK root directory

Solution creation and build process

The preferred way is to open a command prompt inside the installation root of the UDK Python Interface, let's assume c:\UDK\PythonInterface.

```
c:
cd UDK\PythonInterface
```

CMake allows the build directory separated to the source directory, so it's a good idea to do it inside an empty sub-directory:

```
mkdir build
cd build
```

The following code requires an installation of CMake and at least one supported Visual Studio version. If CMake isn't included into the **PATH** environment variable, the path must be specified as well:

```
cmake ..
```

This searches the preferred Visual Studio installation and creates projects for it. Visual Studio Express users may need to use the command prompt offered by their installation. If multiple Visual Studio versions are installed, CMake's command parameter '-G' can be used to specify a special one, see CMake's documentation in this case. This process creates the solution files inside *c:\UDK\PythonInterface\build*. All subsequent tasks can be done in Visual Studio (with the created solution), another invocation of cmake isn't necessary under normal circumstances.

Note: Be sure to build the release version, as the Python debug libraries are not included in the standard Python installation package.

Important: The UDK Python Interface must be build with the same toolchain and build flags like the UDK C++ API. Otherwise CMake will not be able to find the compiled UDK libraries.

Info: It is easy to create different builds with different Visual Studio versions by creating different build directories and invoke CMake with different '-G' options inside them:

```
c:
cd UDK\PythonInterface
mkdir build2005
cd build2005
cmake -G"Visual Studio 8 2005" ..
cd ..
mkdir build2008
cd build2008
cmake -G"Visual Studio 9 2008" ..
```

Linux

Requirements

- Python 3.1.2 → www.python.org

Note: The UDK Python Interface will not work with Python 2 as Python 3 is backwards-incompatible.

The working directory for own applications must contain

• UDK C++ API (built version)	→ libudkapi.so
• Python dynamic link library	→ udk_python_if.so
• Python class definition	→ udk_py_if_class.pyc

To build the interface the following is also required:

- GNU C++ compiler toolchain
- CMake 2.6 or higher → www.cmake.org

```
sudo apt-get install build-essential cmake python3.1-dev
```

Build the UDK Python Interface dynamic library

Prerequisites

There are some options that need to be fixed in *msvc.cmake* inside the UDK Python Interface root directory:

- **PYTHON_INC_DIR** Path to the Python include directory (e.g. /usr/include/python3.1)
- **PYTHON_LIB_DIR** Path to the Python library directory (e.g. /usr/lib/python3.1)
- **CMAKE_BUILD_TYPE** Select build type, can be one of *Debug, Release, RelWithDebInfo, MinSizeRel*. If there should be at least 2 builds in parallel, remove this line and specify the type using command line option *-DCMAKE_BUILD_TYPE=…*.
- **UDK_ROOT** Path to the UDK root directory

Makefile creation and build process

The preferred way is to open a command prompt inside the installation root of the UDK Python Interface, let's assume /UDK/PythonInterface.

```
cd /UDK/PythonInterface
mkdir build
cd build
cmake ..
```

If all external dependencies are met, this will finish creating a Makefile. To build the UDK Python Interface, just invoke make:

```
make
```

Important: The UDK Python Interface must be build with the same toolchain and build flags like the UDK C++ API.

How to use the UDK Python Interface

This example shows how to call the predefined functions of the Python interface class in a Python script.

Import the UDK in your Python script:

```
from udk_py_if_class import ceDevice
```

For this example some design related constants must be defined:

```
EFM01_BRAM_BASEADDR = 0x00000000
EFM01_BRAM_HIGHADDR = 0x000007FF
```

In the main script function the API must be initialized before enumerating and accessing devices. To keep the example simple only all supported EFM01 devices will be enumerated.

```
ceDevice.Init()
ceDevice.Enumerate(ceDevice.ceDT_USB_EFM01)
```

To access the first found EFM01 device, a handle to device number 0 is needed.

```
pDev = ceDevice.GetDevice(0)
```

The device can be prepared for access by calling the Open() function using the received device handle:

```
pDev.Open()
```

Now some device specific information can be retrieved:

```
print("--------------------------------------------------")
print("- Device: " , 0)
busType = pDev.GetBusType()
if(busType == ceDevice.ceBT_PCI):
    print("- Bus: PCI")
elif(busType == ceDevice.ceBT_USB):
    print("- Bus: USB")
else:
    print("- Unknown bus type " )

print("- Device-Type: " , pDev.GetDeviceName())
print("- Device-UID: " , pDev.GetDeviceUID())
```

A FPGA design is needed to configure the FPGA. Here the “efm01_top.bin” design for EFM01 devices is used.

```
pDev.ProgramFPGAFromBIN("efm01_top.bin")
```

After successfully configuring the FPGA some values can be written, read back and compared.

```
uiBaseAddress = EFM01_BRAM_BASEADDR
uiWVal = 0x11111111
uiRVal = 0
i = 0
while i < 4:
    pDev.WriteRegister(uiBaseAddress, uiWVal)
    uiRVal = pDev.ReadRegister(uiBaseAddress)
    if(uiWVal != uiRVal):
        print("data inconsistence.")
    else:
        uiWVal = uiWVal * 2
    i = i+1
```

Finally the device should be closed and the API must be deinitialized to safely free the access to all used devices and resources.

```
pDev.Close()
ceDevice.DeInit()
```

Methods/Functions of the UDK Python Interface class

Note: For more information about the functions have a look at the UDK documentation, as the Python Interface is just a wrapper calling the UDK API functions internally.

Note: The *self* argument in Python functions indicates a reference to the object itself and by convention, it is given the name *self*. Do not give a value for this parameter when you call these methods, Python will provide it.

GetUDKVersionString

Python	GetUDKVersionString()

Return string which contains the UDK version.

Init

Python	Init()

Prepare internal structures, must be the first call to the UDK API. Can be called after invoking DeInit() again.

Enumerate

Python	Enumerate(uiDeviceType)

Search for (newly plugged) devices of the given type and add them to the internal list. Access to this list is given by GetDeviceCount() / GetDevice().

DeviceType can be one of the following:

DeviceType	**Enumerate(Description)**
ceDevice.ceDT_ALL	All UDK supported devices.
ceDevice.ceDT_PCI_ALL	All UDK supported devices on PCI bus.
ceDevice.ceDT_PCI_PCIS3BASE	Cesys PCIS3Base
ceDevice.ceDT_PCI_DOB	DOB (*)
ceDevice.ceDT_PCI_PCIEV4BASE	Cesys PCIeV4Base
ceDevice.ceDT_PCI_RTC	RTC (*)
ceDevice.ceDT_PCI_PSS	PSS (*)

ceDevice.ceDT_PCI_DEFLECTOR	Deflector (*)
ceDevice.ceDT_USB_ALL	All UDK supported devices.
ceDevice.ceDT_USB_USBV4F	Cesys USBV4F
ceDevice.ceDT_USB_EFM01	Cesys EFM01
ceDevice.ceDT_USB_MISS2	MISS2 (*)
ceDevice.ceDT_USB_CID	CID (*)
ceDevice.ceDT_USB_USBS6	Cesys USBS6

* Customer specific devices.

DeInit

Python	DeInit()

Free up all internal allocated data, there must no subsequent call to the UDK API after this call, except Init() is called again. All retrieved device pointers and handles are invalid after this point.

GetDeviceCount

Python	GetDeviceCount()

Return count of devices enumerated up to this point. May be larger if rechecked after calling Enumerate() in between.

GetDevice

Python	GetDevice(uiIdx)

Get device pointer or handle to the device with the given index, which must be smaller than the device count returned by GetDeviceCount(). This pointer or handle is valid up to the point DeInit() is called.

Open

Python	Open(self)

Gain access to the specific device.

Close

Python	Close(self)

Finish working with the given device.

ReadRegister

Python	ReadRegister(self, uiRegAddr)

Read 32 bit value from FPGA design address space (internally just calling ReadBlock() with size = 4).

WriteRegister

Python	WriteRegister(self, uiRegAddr, uiValue)

Write 32 bit value to FPGA design address space (internally just calling WriteBlock() with size = 4).

ReadBlock

Python	ReadBlock(self, uiAddress, uiLen, bIncAddr)

Read a block of data to the host buffer which must be large enough to hold it. The size should never exceed the value retrieved by GetMaxTransferSize() for the specific device. bIncAddress is at the moment available for USB devices only. It flags to read all data from the same address instead of starting at it.

WriteBlock

Python	WriteBlock(self, uiAddress, ucData, bIncAddr)

Transfer a given block of data to the 32 bit bus system address uiAddress. The size should never exceed the value retrieved by GetMaxTransferSize() for the specific device. bIncAddress is at the moment available for USB devices only. It flags to write all data to the same address instead of starting at it.

WaitForInterrupt

Python	WaitForInterrupt(self, uiTimeOut)

(PCI only) Check if the interrupt is raised by the FPGA design. If this is done in the time specified by the timeout, the function returns immediately flagging the interrupt is raised (return code / *puiRaised). Otherwise, the function returns after the timeout without signaling.

Important: If an interrupt is caught, EnableInterrupt() must be called again before checking for the next. Besides that, the FPGA must be informed to lower the interrupt line in any way.

EnableInterrupt

Python	EnableInterrupt(self)

(PCI only) Must be called in front of calling WaitForInterrupt() and every time an interrupt is caught and should be checked again.

ResetFPGA

Python	ResetFPGA(self)

Pulses the FPGA reset line for a short time. This should be used to sync the FPGA design with the host side peripherals.

ProgramFPGAFromBIN

Python	ProgramFPGAFromBIN(self, ccFile)

Program the FPGA with the Xilinx tools .bin file indicated by the filename parameter. Calls ResetFPGA() subsequently.

ProgramFPGAFromMemory

Python	ProgramFPGAFromMemory(self, ucBuffer, uiSize)

Program FPGA with a given array created with UDKLab.

ProgramFPGAFromMemoryZ

Python	ProgramFPGAFromMemoryZ(self, ucBuffer, uiSize)

Same as ProgramFPGAFromMemory(), except the design data is compressed.

SetTimeOut

Python	SetTimeOut(self, uiTimeOut)

Set the timeout in milliseconds for data transfers. If a transfer is not completed inside this timeframe, the API generates a timeout error.

EnableBurst

Python	EnableBurst(self, bEnable)

(PCI only) Enable bursting in transfer, which frees the shared address / data bus between PCI(e) chip and FPGA by putting addresses on the bus frequently only.

GetDeviceUID

Python	GetDeviceUID(self)

Return string formatted unique device identifier. This identifier is in the form of *type@location* while type is the type of the device (i.e. *EFM01*) and location is the position the device is plugged to. For PCI devices, this is a combination of bus, slot and function (PCI bus related values) and for USB devices a path from device to root hub, containing the port of all used hubs. So after re-enumeration or reboot, devices on the same machine can be identified exactly.

GetDeviceName

Python	GetDeviceName(self)

Return device type name of given device pointer or handle.

GetBusType

Python	GetBusType(self)

Return type of bus a device is bound to, can be any of the following:

Constant	Bus
ceDevice.ceBT_PCI	PCI Bus
ceDevice.ceBT_USB	USB Bus

GetMaxTransferSize

Python	GetMaxTransferSize(self)

Return count of bytes that represents the maximum in one transaction, larger transfers must be split by the API user.

Table of Contents