The netpp HOWTO

Martin Strubel

February 28, 2015

Revision:
`0.5x-develop`

Why another library?

Make embedded devices talk to each other and tell each other what they are capable of

The early ideas about a framework to master the recurring task of software interfacing with embedded devices came up in 2003. Quite a number of solutions were on the market at this time, but none of them as either abstract enough or leave such a tiny footprint that it would run efficiently and almost unnoticed on an embedded system or on the front end PC itself. After a few years of refining and withstanding the requirements of scalability and robustness in industrial applications, the common and generic framework is ready to be released as OpenSource.
Originally, the first rewrite started out as **DClib** - for device control library. This toolset allowed to describe hardware entities of a device (registers, bits, etc.) in a newly defined XML dialect with a graphical editor and generate communication libraries from it. Those communication libraries however, were very protocol specific, the low level physical layer was still running each device's proprietary communication procotol.
With emerging networked and other well standardized physical communication capabilities of various embedded devices, the need for a unified client software came up - a software module, that would allow to communicate with all devices that are somewhat aware of their properties. Basically this meant a transfer of the client side DClib based communication libraries down to the embedded target and creation of a new protocol: The network property protocol **netpp**.
There are various approaches to netpp, depending on whether you are a hardware developer, firmware programmer, high level user interface programmer or project manager. This HOWTO only gives you an overview on the mid to high level design layers of the netpp library. For the programming details, the API documentation is most up to date. The author hopes, that this document helps you to find an easy entry point into netpp. For all other documents covering the different approaches, please see the main netpp resource site [netppres] Appendix B.1.
In more recent revisions of this document, some more low level details on hardware design, in particular covering VHDL, was added. Meaning, that the device description language is capable of automatically generating hardware description language as well. This makes the following work flow easy:

- Design a system on an FPGA.
- Create drivers according to automatically generated headers.
- Access the implemented FPGA peripherals using any register protocol or even netpp.

1 Getting started

1.1 Installing and compiling netpp

In the good tradition of OpenSource, we assume that you are a programmer and somewhat familiar with Linux and various command line tools

netpp is heavily depending on the 'make' tool (in particular the Gnu variant). To automatize the building process, it is very much recommended to use Makefiles, as demonstrated in the `devices/example` folder of the netpp distribution.

It is also possible with other tools (for example the Microsoft Visual C++ tools) to automatically build source files using custom rules. However, this technique is not covered here.
Although netpp runs on many platforms and various operating systems, its home truely is a linux environment. For Microsoft Windows, there are various Unix-like environments such as MinGW or Cygwin that netpp can be compiled under. Also it is possible and very convenient to cross-compile for Windows and the embedded target on a Linux host with one 'make' call. Again, this technique is not covered in this HOWTO.
To install and compile netpp, obtain the netpp distribution tar file via http://www.section5.ch/netpp and extract it using:

```
tar xfz netpp_dist.tar.gz
```

– or use your favorite archive manager.
Then you enter the created folder netpp and run **make**. If all goes well, you should end up with a number of executables, the most important being:

master/master
: The generic netpp client

devices/example/slave
: An example device slave

See Section 1.2 how these examples are used.
If the compilation fails, see next section about prerequisites.

1.1.1 Prerequisites

- xsltproc
- GCC, Microsoft Visual C++, Borland C
- Python developer distribution (tested: v2.4-v2.7). Python 3.0 is not yet supported.

On a Linux system, these packages are normally installed using a package manager and come with almost every distribution. On Windows, it is recommended to install the Cygwin or MinGW/MSys distributions. They offer a large selection of various tools plus a decent command line shell.

1.2 Testing the example

After successful compilation, you can test the example on your local host machine. Enter the folder `devices/example/` and run the slave executable by typing **./slave**
You should then see the following output on the slave console:

```
ProbeServer listening on UDP Port 7208...
Listening on UDP Port 2008...
Listening on TCP Port 2008...
```

Open up another console window and enter the `master/` folder. Now run the master tool by typing **./master**
If the network is set up well, a probe request will be sent out and answered by all running slaves. Each slave responds by listing its available ports and services.

 The probe request will be only answered if the target supports UDP.

You can now address a specific slave using

```
./master TCP:localhost
```

You will then get the property list as follows:

```
Protocol Version 1
Checksum: 0426
Properties of Device 'DerivedDevice' associated with Hub 'TCP':
Child: [10000001] 'Demo'
Child: [10000002] 'Extra'
Child: [10000003] 'Test2'
Child: [00000002] 'ControlReg'
Child: [00000003] 'ControlRegH'
Child: [00000004] 'ControlRegL'
Child: [00000009] 'FloatVal'
Child: [0000000c] 'Integer1'
Child: [00000012] 'Mode'
Child: [00000016] 'StaticString'
Child: [00000017] 'StaticTest2'
Child: [0000001a] 'VariableString'
Child: [00000020] 'Zoom'
Child: [0000000d] 'LED'
Child: [00000019] 'Stream'
Child: [0000000e] 'LUTarray'
Child: [0000000f] 'LUTvalues'
```

Now query a property, e.g. Mode:

```
./master TCP:localhost Mode
```

You will now see its children. To browse through the tree structure, use the typical dot notation as in various object oriented languages:

```
./master TCP:localhost Mode.Fast
```

This is the value for the 'Fast' operation mode. If you now wish to change the mode, try setting it to this value:

```
./master TCP:localhost Mode 1
```

A movie on how the example can be addressed using various methods (console, scripting, graphical user interface) is demonstrated in the presentation found at http://www.section5.ch/netppres.

When installing a prebuilt netpp package for Debian/Ubuntu, the 'master' tool is just called netpp.

1.3 Using the XML editor

For XML file editing, netpp provides several auxiliary files found in the `xml/` subdirectory. Most XML editors will be able to process XSD files, however it is recommended to use the XMLMind XML Editor (XXE) [xxe] Appendix B.1.
The most relevant support files for graphical XML editing using XXE are:

- devdesc.xsd, interfaces.xsd: The device description schema. This defines our device description language
- devdesc.css: A cascading style sheet for the property editor
- devdesc.xxe: The addon configuration for the XXE

Before creating new XML device descriptions, it is recommended to install these files as XXE addon. For this, it is sufficient to just link the entire `xml/` directory to the `addon/` directory of the XXE distribution, giving the name `devdesc`. Alternatively, you can use `$HOME/.xxe/addon` instead. For Windows operating systems, please check the detailed instructions at http://www.xmlmind.com/xmleditor/addons.shtml#manual_install.
For example, under linux use the command:

```
ln -s ~/src/netpp/xml ~/.xxe/addon/devdesc
```

For XXE versions > v5.x.x, use the ~/.xxe5 directory.

Once you have successfully installed the netpp addon, you will see the DCLIB/NETPP template option when creating a new file via the **File**->**New** menu, as shown in Fig. 1.1.
Sometimes, it can be necessary to manually modify the XML file, for example when the netpp directory is moved elsewhere, or a relative path to the XSD file is used. In this case, open up the XML file with a programmer text editor and look for the following line

```
xsi:schemaLocation="http://www.section5.ch/dclib/schema/devdesc
   ../../xml/devdesc.xsd"
```

If there is a relative path specified like above, and XXE complains it can not find the schema, change the path accordingly.
The sample may apply to the style sheet devdesc.css, when it is not found. In this case, you have to modify the line

```
<?xml-stylesheet type="text/css" href="../../xml/devdesc.css"?>
```

You can also change paths inside the XML editor, please consult the XML editor documentation for detailed instructions.

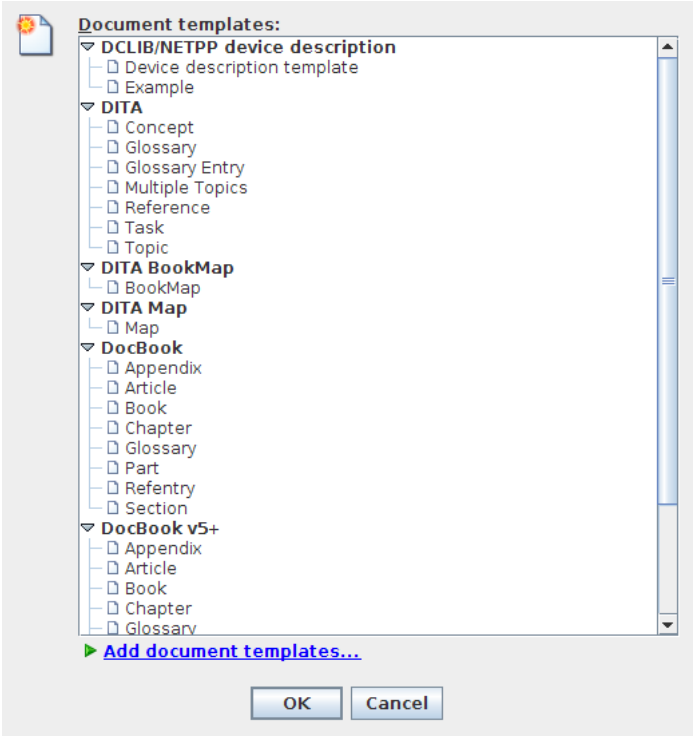

Figure 1.1: Template choice with installed netpp addon

Another method to specify or translate search locations, are XML catalogs. See also `xml/devdesc_catalog.xml` of the netpp distribution for example.
When using the addon installation method from above, the default catalog points to the files in `xml/` of the netpp distribution.

1.4 XML Translation

All auxiliary files to translate XML into various target formats are contained in the `xml/` folder of the netpp distribution.
Table 1.1 shows the default XSLT stylesheets coming with netpp. Some of them have a default make rule, this means, that there is a defined rule in `xml/prophandler.mk` that creates the target file from the device file. For example, **make register.h** will extract the register definitions from the XML device file and create a C header file. Some files that are part of internal netpp dependencies are always created automatically.
Most of these XLSTs are called with specific parameters. For details, see the XSLT reference in Chapter 5.
The auxiliary `prophandler.mk` file is typically included from a project makefile and needs two variables set:

NETPP
: An absolute or relative path to the netpp top level directory

DEVICEFILE
: The full name of the device file (should be in the local directory)

When including `prophandler.mk` from a separate project, it makes sense to use **absolute** paths for the `$(NETPP)` variable.

XSLT file	Default make rule	Description
registermap.xsl	-	Common auxiliary translation sheet for inclusion
registerdefs.xsl	register.h	Legacy: Creates register definitions without bitfields
reg8051.xsl	-	Creates register definitions header with bitfields. Note: registermap with id=SFR is treated specially (for 8051 compatible CPUs)
proplist.xsl	proplist.c	Creates the C source code property list
userhandler.xsl	handler_skeleton.c	Creates a function handler skeleton
proptable.xsl	device_properties.xml	Creates register and property documentation
regwrap.xsl	-	Creates direct register wrapping Properties from register definition
errorhandler.xsl/errorlist.xsl	(automatic)	Creates generic error handler and error code header file. Newer netpp versions call this with a special parameter to create custom error handling targets. See Section 2.1.

Table 1.1: XML default style sheets

2

Creating device properties

To understand the nature of a Property, a few basics will have to be introduced. A Property is actually a named entity, but to avoid having to deal with strings all the time, it is actually addressed via a TOKEN. A TOKEN is just a 32 bit value, which can be arbitrarely encoded on the target. When speaking to a device and addressing a specific property by name, the device normally reports its token. A well matching analogy is the domain name lookup in the internet, you get an IP number from the name server, and then you actually start communicating. More about the TOKEN internals are found again in the API documentation [apiref] Appendix B.1. For now, we'll only mention the TOKEN when it gets to technical details. If you wish to work down from the top to the bottom and do an abstract feature set design first, you can safely ignore the technical layer and start with the XML authoring only.
Opposed to the rather abstract nature of a property is the actual hardware definition (which is optional). A hardware definition is required when you are actually accessing a low level peripheral. In the most field applications, the direct access of register properties is often not desired due to stability criterias, as guarding software routines might be missing to prevent the writing of illegal value combinations. However, with growing intelligence of devices and clever designs, an increasing number of peripherals already provides the safety for direct access, moreover, direct access is always a useful strategy for a device debugging mode or the prototyping/testing phase of a development.
Probably the most important subject is how a system reacts to errors. These can occur all the time when using remote communication features. Therefore we will discuss netpp's error handling strategy first

2.1 Error handling

In general, netpp functions return an error code. Likewise, all embedded code must communicate the failures to the netpp protocol wrapper, i.e. must return a code. The return codes are differentiated as follows:

- <0 : Error (failure of a certain severity, depending on the code)
- >0: Warning (no failure, but attention required)
- 0: No error, all ok.

The error/warning return codes are actually defined in XML. Here, we have to differentiate again:

- netpp native codes: errors and warnings that are intrinsic to the netpp framework
- device specific codes: error codes that are non-generic and specific to a certain device implementation

All device specific errors are defined in the device description XML file.
By default, a netpp client (master) only knows the netpp native error code descriptions, but not the device specific ones. To retrieve these error descriptions from the target, some kind of lookup dictionary (a 'dict type node') will have to be emulated. The proposed, but not yet standardized way to do this is via the reserved `ReturnCodes` Property. See `devices/example/` folder how this is done. Basically, the error handling extension is implemented as follows:

1. Implement a `ReturnCodes` Property (can be hidden) on the top level of your device description according to the group template "ErrorHandling" in `example.xml`
2. Implement the handler functions `set_retcode_code()`, `get_retcode_code()` and `get_retcode_description()` according to the templates in `handler.c`
3. Create a device specific error handler using the `errorhandler.xsl` stylesheet by running **make errhandler.c**. This implements a function `device_strerr()` and a code table which is referenced by the above handler functions
4. Iterate through the ReturnCodes dictionary emulation by:
   (a) Set `i = 0`
   (b) Set ReturnCodes.Code property value to `i`
   (c) Get ReturnCodes.Code property value to retrieve the return code, stop when this return code is = 0
   (d) Retrieve description of this code via the STRING property ReturnCodes.Desc
   (e) Increment `i` and repeat from (b)

The netpp command line master tool implements the above method. Note: The ReturnCodes property is now implemented as struct type property. In future, it may be a dedicated dynamic property (dictionary type property).

2.2 Atomic Property types and their purpose

This section describes the various property types and their typical use cases. For technical and implementation details, please refer to the API reference.

2.2.1 BOOL - Boolean, or a single bit

A boolean – the most atomic property – is normally used to address single bits within registers. However it can also refer to a state or a two state operation mode (on or off). Typically, an Enable property is of type BOOL.

2.2.2 INT - A 32 bit signed integer

An INT is just an integer number in the entire range of a 32 bit representation. This is normally enough for a typical device configuration. If you need to transfer a specific byte width, use a BUFFER type. Integer property nodes can have a Max and a Min member to denote valid boundaries. If these boundaries are exceeded, an error will be returned by the built-in property handler. Boundaries can also be dynamic, please refer to the API documentation for details. A boundary node is just another property, so it does have the same attributes. For dynamic handling of propertes, see Section 2.5.

2.2.3 MODE - An operation mode

A mode is typically an operation mode of a device. For example, you may have a video display that is capable of two orientation modes (Portrait, Landscape) and a few video formats. On a hardware level, this is normally implemented as a register bitfield where you write a certain value out of a set of valid modes. These are represented by a `choice` node which again has a list of valid items.

2.2.4 FLOAT - A 32 bit IEEE float type

A FLOAT is just a 32 bit wide C type float value. Normally, you would have some kind of routine to convert a float number into a valid register integer value or bit combination. Therefore, a FLOAT property can not use a register reference (see Section 2.7.3)

2.2.5 STRING - A null terminated character array

A STRING type property is normally used to pass named cleartext strings around. However, you are also free to use a specific encoding.

2.2.6 BUFFER - Arbitrary data blocks with defined size

Arbitrary buffer data can be passed between netpp capable clients and servers using a BUFFER property. For example, an image frame or audio data would be transmitted using a BUFFER property. Bulk array data can also be packed into a buffer. A possible netpp enhancement would be to describe the data contained in a buffer as bulk packet to contain several properties at once. This is projected for the netpp 1.0 version.
Buffer handling can be implemented "zero copy" on some platforms, such that no data is mirrored at extra overhead. For details, see Section 2.8.4.

2.2.7 COMMAND - An action to be executed on the target

A COMMAND property mostly represents in a button in the graphical remote control front end. The actual value of a command normally does not matter. It is not recommended to use different command property values to execute command options, except for debugging purposes. Writing of a command property normally executes the action on the target, reading the command property returns the status whether the command has executed (0), a positive BUSY value (own warning codes can be implemented), or a negative error code.

2.2.8 REGISTER - A specific register property

This is treated the same as an INT, but without sign. This property is used to denote the access of a raw register. However, an INT can be used safely instead. Normally, a REGISTER property is only used for prototyping or debug mode and not exported to the netpp user.

2.3 Non-Atomic Properties

Properties can also be of a container type, such as a `struct` or `array` node. These are instanced as a STRUCT respectively ARRAY meta type within the property list and have other properties as children. A struct node is basically just a namespace container. An array note is much more powerful: it can be used to define a repetition of entities with a normally fixed size. An array property is normally introduced, when entire register tables (such as mapped memories) need to be mapped and accessed by element instead of using a BUFFER transfer. The typical scenarios:

- A low level peripheral, for example a sensor, has a built-in configurable look up table for sampled values
- The device has several operation contexts which can be switched to using one variable (e.g. an INTEGER or MODE property), but use separate configurations for these contexts

ARRAY Properties can contain STRUCT Properties, but currently not another ARRAY. Therefore, multidimensional arrays are not supported in the current implementation.

2.4 Special property attributes

- RO/WO: Read only, write only
- VOLATILE: Property can change. Normally, these are properties that are to be read out repeatedly for status display
- HIDDEN: This property is not shown in the property hierarchy list. It can only be accessed if its name or TOKEN is known.

2.5 Property access (Handlers, Variables, Registers)

Basically, a property can be implemented in the following ways:

1. As a static value, read-only
2. As a variable that is accessed through a global structure or context
3. As a hardware-implemented register or bit vector
4. As a handler routine set via a getter and a setter (depending on read/write access)

Table 2.1 gives an overview about the possible access method children nodes of a property.

Access method/implementation	XML node name	Typical use case
Static value, read-only	value	A constant identification value or name
Globally accessible variable	variable	A configuration value or operation parameter, implemented in a global configuration option structure
Hardware-Register or bit vector	regref	A mapping of a hardware entity into a named Property
Getter or setter routine	handler	More complex accesses or manipulation of operation states that involve extra actions, or Properties that implement command sequences ('Start'/'Stop')

Table 2.1: Access method node lookup table

A variable or handler access node requires an implementation in C source. See API reference for details. A static value is encoded directly in the property list. A register again is assumed to be declared prior to the actual property definition. This will be elaborated in the next section Section 2.7.

2.6 Dynamic properties

In some rare cases, a device might know its properties from another definition than a static hardware description XML or generate several number of property instances automatically. For example, a device that has a number of hot-pluggable slave devices attached may want to enumerate its 'dumb' clients and notify the netpp client user.
An existing case are the internal Port nodes of a Hub: On a device probe (broadcast), all attached devices will respond and be shown as a Port property that can be connected to.

Since properties are organized in a tree structure, auxiliary functions are supplied to build a dynamic property device root node with children. Internally, they have a different structure from the statically built property table. Each dynamic property is accessed by a standard TOKEN of a specific type 'DynPropToken'.
To construct the tree, a dynamical property ('DynProp') first needs to be allocated and then inserted into the hierarchy. Building the DynProp occurs using the `new_dynprop()` function using a Property Descriptor template. This again is a static descriptor like found in the generated property list (`proplist.c`).
The following example demonstrates how to build a simple dynamic property tree:

```
/** A root node template */
PropertyDesc s_rootprop = {
    .type = DC_ROOT,
    .flags = F_RO | F_LINK /* Derived ! */,
    .access = { .base = 0 }
};

/** A buffer Property Description template */
PropertyDesc s_buffer_template = {
    .type = DC_BUFFER,
    .flags = F_RW,
    .where = DC_CUSTOM,
    .access = { .custom = { buffer_handler, 0 } },
};

TOKEN prop_builder(void)
{
    TOKEN root, t;

    dynprop_init(80); // Reserve space for 80 dynamic properties

    root = new_dynprop("DynPropDevice", &s_rootprop); // assume this never fails
    t = new_dynprop("Buffer", &s_buffer_template);
    if (t == TOKEN_INVALID) return t;
    dynprop_append(root, t);  // Insert Buffer property as first root node child
    return root;
}
```

This example uses class derivation to support a static AND dynamic property definition at the same time. See also Section 4.3.2.

Using recursive calls and the above functions, an entire statically defined property hierarchy can be cloned as a dynamical instance. Note though that removal of dynamical properties from the hierarchy is only supported in netpp v1.0.
Please see API documentation about details on usage of these functions.

The evaluation version of netpp may not support dynamic properties

2.7 Hardware definition

When implementing access to a peripheral device of an embedded system or even designing a new peripheral, it can possibly be agreed on that the simplest approach is the register based one. A register is merely defined as an adressable value of a certain bit width that can be read or written by an interface using a specific low level or hardware protocol. Another differentiation is made between protocols with synchronous and asynchronous attribues, however this is not the scope of this HOWTO. What you have to know as a developer, is, that you have to supply the low level register access methods by a `device_read()` and `device_write()` function. These **peripheral handlers** call hardware protocol library functions or actually implement the procotol. The interface to these functions is memory based, i.e. specifies an address parameter and the actual data block of a certain size. Since the register bit width can be variable, it is up to the peripheral handlers to properly decode the values from the data block and size.

2.7.1 Address decoding: Peripheral handlers

As you have only the peripheral handler low level API which represents access through one big linear address space, but possibly a number of peripherals attached via various bus systems or interfaces, you will have to sort out your own address decoding. Basically, the definition of several peripherals works as follows:

1. Create a registermap node for each peripheral
2. Inside the registermap, create the register definition nodes for each register needed. Use the peripheral's local, relative addressing for each register, just as it is defined in the hardware reference
3. Depending on each peripheral's address space, introduce a virtual address offset for each registermap such that you can for example determine the hardware bus interface using a bit in the so created virtual I/O space, another number of bits for the device's address on that bus, etc.

Inside the peripheral handler, you can then do the address decoding according to those virtual address bit values.
Another aspect is the following: A device can be byte-addressable like a typical memory, but also word-addressable, like on many embedded architectures. The `addrsize` attribute of the registermap node defines, how many bits are accessed per address value. This attribute specification is however optional and only used for code generation on special architectures. Normally, you would take care of the word width inside the peripheral handler. So you have to sort out the block size and the appropriate conversion based on the parameters to the peripheral handler.

To communicate an illegal register size (e.g. size 1 on a 16 bit address bus) or illegal address to the caller, return a DCERR_COMM_FRAME error code from `device_read()` or `device_write()`.

2.7.2 Registers and Endians

There are many ways to Rome. This also represents in the various implementation of registers, Endianness being one of the biggest issues when it comes to registers having a greater size than eight bits: It's the matter of transfering the low byte (little endian) or the high byte (big endian) first. Or just the order that multi byte values have in internal memory.

You, as a programmer, may have to sort out different peripherals with different endianness. The DClib framework offers you a few features to reduce the programming overhead, but there are again several ways how to do it. We will just look at a few scenarios:

Different Endian devices on one bus
Use the `endian` attribute of the `registermap` node

Mixed endians on different interfaces, not sharing a bus
Possibly implement the necessary endian swapping in the peripheral handlers and maintain a uniform endianness over the register maps

One important property of a register node is the `size` attribute. This is the registers effective size **in bytes**. Even if you could change the `device_write()` and `device_read()` handlers to deal with different size units, say in bits, the netpp client and protocol always deals with bytes. If you wish to specify a different internal addressable width, you have to use the `addrsize` property of the parenting register map. This is typically only necessary for deriving a customized hardware definition via your own translation style sheet (Section 6.2).

2.7.3 Register to property mapping

There are various in depth tutorials found at the netpp resource home page [netppres] Appendix B.1, but basically, the mapping works such that you simply create a `regref` node inside a Property definition, containing a reference to the `register` node's ID. This accesses the entire register, if you wish to access partial bit vectors of a register only, you have to specify the name of the bitfield defined inside the referenced register in the `bits` attribute. This way you can for example map a single register bit into a boolean Property. The currently supported type mappings for register entities are INT, REGISTER and BOOL. Float to register mappings have to be encoded using a property handler.
For prototyping or just for generating a skeleton, automated Property instances of registers can be created using the method in Section 4.1.

2.8 Design guides

There are many possible approaches and methods to a XML based device design. A hardware engineer will think in terms of logic such as bits, bytes and registers, a user interface specialist will rather focus on simplicity and least confusion for a non expert user. It may be adviseable to start with a top to bottom design approach, but this is not always possible. Therefore, we will again present a list of possible ways in the following sections.

2.8.1 Naming

The device description language is pretty relaxed concerning variable naming standards. The naming of most entities accords to the XML id standard which forbids some special characters only. This is one prerequisite to be able to turn XML into valid C code. However, since our XML dialect introduces as little restrictions as possible, bad C code can possibly be generated by improper naming. Therefore keep in mind: C/C++ style naming is the safest way to go.
For the naming scheme of entities, you are pretty free in using your personal naming conventions. However, we list a set of recommendations and short reasons in Table 2.2.
When talking about naming, we also have to mention the name spaces in our XML dialect. As you could guess, an ID must be unique, i.e. a specific `id` attribute name of an XML node can only occur once in the entire XML file. The XML editor will normally warn you if you have multiple entries of the same ID.
Register definitions and device nodes for example should be unique for obvious reasons. However, since bitfield nodes are never directly referenced but addressed by their parenting

XML node	Example	Description	Reason
property	ExposureTime	C++ style, avoid underscores	It's a named thing, exposed to the user. Keep it readable.
register	Status_Control	C++ style or capitals	Will turn out as internal #define, readability not an issue. A 'Reg_' prefix will be prepended. Is a unique ID
bitfield	AUTO_ENABLE	Capital style	Not a unique ID, turns out as internal #define
(Other nodes)	simple_camera	lower caps C style	Just a normal ID for internal reference

Table 2.2: Naming recommendations

Reset	Resets the entire device
Enable	Enables the device (wakeup from standby)
PowerDown	Turn off the device (if allowed)
Mode	The global operation mode of the device

Table 2.3: Standard top level properties

register definition first, they do not need to have unique names. For example, you could have an `ENABLE` bit in a `Control` register, but also in a special `PowerSave` register. This can be a problem when generating a custom C header for direct access from your own routines, as there will be conflicting `#define` statements. The C preprocessor will normally warn you about redefinition of symbols. To avoid name clashes, adaptation of XSLT files is recommended such that those symbol names are unique, for example by prepending the parenting register ID. It is pretty much the developer's choice, whether an approach for well-readability or namespace-safety should be taken.

Reserved properties

For historical, but also for future compatibility reasons, some property names are reserved for special purposes. They are listed below in Table 2.4.

2.8.2 Usage scenarios

Before starting to implement a netpp application, it is wise to think through the desired roles of the concerning remote device components. Also, the design may be limited to a certain scenario, depending on the physical communication layer. For example, a standard, one-endpoint USB interface has a clear master/slave relationship by default, so the USB Host will always be the master. However, this role can - under certain prerequisites - be reversed, for example, if the USB device offers a second end point. Since this is rather hardware specific, specific USB drivers (Hubs) must be implemented in netpp to satisfy the need of a 'slave' channel that enables the Host to react to attention messages.
For now however, we will assume that the solution can be simpler, boiling down to a few use cases, where the actual device control and configuration action is:

DeviceClass	Defines a registered device class with minimum set of properties
XRCFile	The XML GUI resource file of a device
Vendor	Struct to hold all vendor specific extensions
ReturnCodes	A dictionary node (netpp 1.0 spec) holding all device specific return codes and their corresponding description strings. In current implementations, this is emulated via handlers, see Section 2.1.
CharEncoding	Character encoding to be used for strings. Not yet defined.
XMLResource	XML resource file for configuration GUI
XMLDialog	Main dialog name
XMLVendorURL	URL to download the above XMLResource from, if not found. Can be a list of ';' separated URLs.

Table 2.4: Reserved property names

- GUI/User driven
- Automated, program or script driven
- Event driven (by device inputs, or alarm conditions)

For the User and automated approach, the front end is always a master, meaning, that the current status of the peer device (slave) has to be polled, i.e. a specific status Property has to be queried repeatedly from an GUI event loop (or script loop).
If the front end does not provide much command action but is rather of the monitoring nature - like a remote display - it is preferrably implemented as slave.
In the networked world of UDP and TCP, the master/slave roles can be easily reversed, an asynchronous two-way communication is easily implemented using threads or a main loop that calls the netpp protocol stack sequentially in slave mode and master mode.
Also, the property approach should be thought through. Let us look at the following scenarios:

1. You are starting completely from scratch
2. You already have an existing static register map of peripherals and want to prototype
3. You have a I/O library that you want to 'property wrap'. You would therefore create handlers.

These strategies are described in the next paragraphs.

Fresh design

A short overview will be given here how things can be done (but don't have to):

1. For each chip you have, insert a `registermap` node into the XML file and add register elements. Make sure to use **relative** addresses for the register specifications, not absolute I/O mappings. The offset address of the register bank should be specified in the offset attribute of the `registermap` node.
2. In the device_read() and `device_write()` functions (peripheral handlers), you always receive an absolute address, thus you have to do the address decoding for your peripherals there and use the appropriate interfaces for the access.
3. Move on to the next paragraph

Existing register map

Assume you have a register map definition ready, but no properties declared yet. For prototyping, you might want to play with registers directly. Having to hand code the properties would be too much hassle in the first place, especially when there are many registers. Therefore, an auxiliary style sheet `regwrap.xsl` can be used to create properties from registers. See Section 4.1 how this works in detail.

Creating handlers

You do have an existing I/O library with getter/setter functions or another property API with some guarding functionality. First, you have to differentiate, whether the function you desire to wrap rather has the nature of a command or a property. If you execute some action on the target that may take a longer time, it is pretty clear that you'd have to wrap this into a COMMAND type property.
If you have a getter and/or setter, they are likely to be wrapped into a value style property. They might also have read only or write only attributes set, depending on existance of getter and setter.

A property handler must return within the protocol timeout, otherwise the protocol layer will report an error. If you dwell in a function for longer, do all command executions in the main loop or another thread and check whether the command has completed by reading the COMMAND type property. This is also described in the API reference.

Eventually, you might up with doing most settings by handlers, to catch forbidden settings under constraints that are imposed by other properties. Depending on your device architecture and complexity, it may be easier writing a handler instead of sorting out valid operation configurations on the hardware.

2.8.3 Property change events

When you change a property on a device, it may very often be the case that the configuration you chose affects the value of another property. Or a specific option could render a number of Properties void or disabled. In the sense of a intuitive user interface, you would want to communicate this change to the user.
Since the standard feedback from a slave is happening via error codes, this scenario is pretty much covered the following way:

- If a property setting was accepted, but had affected another setting, a `DCWARN_PROPERTY_MODIFIED` is returned from a `dcDevice_SetProperty()` call.
- To determine, what has changed, the Event type children of the property should be queried. See API documentation for details.
- If the property has no Event children, it can be assumed that it has changed itself.

Typically, you re-read the properties that have changed according to the Event children list (which consists of property tokens).

Only handle a DCWARN_PROPERTY_MODIFIED reread action in the context of a `dcDevice_SetProperty()` call. Otherwise you may possibly end up in an endless 'refresh' loop, if the target library responds with faulty error codes.

A complex device design in conjunction with graphical user interfaces is out of scope to be fully covered by this HOWTO. For detailed insight, you may have to contact the netpp authors.

2.8.4 Data transmission using buffer queues

netpp was designed with the least possible buffer copying overhead in mind. Therefore the buffer handling may seem a little complicated. One principle that a programmer should never forget:

Always be sure who owns and currently uses the buffer

Thinking in netpp properties, one has to know whether a buffer is **dynamic** or **static** on the slave side. To elaborate: A static buffer is normally a statically declared memory range in the targets program data section with a fixed size. For example, a version information string is always static. A dynamic buffer can be allocated on the target ad-hoc, it may have a fixed or dynamic size, depending on the implementation.
A buffer property is - when not static - always handled via the handler method, i.e. getters and setters, living normally in `handler.c`. On a buffer transaction, these handler routines are always called **twice**. On the first call, the DCvalue `type` field is set to the data type, i.e. `DC_BUFFER` or `DC_STRING` and expected to return a pointer to a valid buffer in the `value.p` member. Also, to prevent buffer overflow, it must return the correct size of the allocated memory buffer in the `len` member.

The buffer can at this point not be altered or freed, because netpp is now transfering the data

On the second call to the handler, the buffer is normally released (if inside a getter) or updated (when inside a setter). On the second call, the `type` field is set to `DC_COMMAND` by the netpp engine. A typical buffer handler therefore looks like:

```
int set_buffer(DEVICE d, DCValue *in)
{
    switch (in->type) {
        case DC_COMMAND:  // This is a buffer update action
            // TODO: Fill in update code!
            break;
        case DC_BUFFER:
            // You must do a buffer size check here:
            netpp_log(DCLOG_VERBOSE, "Set buffer len %d", in->len);
            if (in->len > BUFSIZE) {
                in->len = BUFSIZE;
                return DCERR_PROPERTY_SIZE_MATCH; // Report to client that it sent too much data
            }

            // Tell engine where the data will go to:
            in->value.p = g_globals.buffer;
            break;
        default:
            return DCERR_PROPERTY_TYPE_MATCH;
    }
    return 0;
}
```

If you plan to allocate buffers on the fly, you can do that just before you initialize the `in->value.p` pointer and process and release/free the buffer inside the `DC_COMMAND` case

handler. However note that your processing time does not exceed the protocol's timeout, otherwise your client will not get the response in time and complain.

A buffer can not be allocated locally on the stack or your program will crash!

Means, the following handler code is strictly forbidden:

```
int set_buffer_NEVER_EVER(DEVICE d, DCValue *in)
{
    char str[32];

    in->value.p = str;
    in->len = sizeof(str);
    return 0;
}
```

Ponder again if the reason for this is not clear: The 'str' variable only exists on the local stack and the netpp engine will transfer the data for you, after you actually left this function. So 'str' does no longer exist!
The above example deals with a fixed size buffer (i.e. static), but there are many more scenarios you can cover using a handler, with respect to flexible buffer sizes:

- When writing:
  - Allocate memory on the fly, according to buffersize expected by client
  - Accept less data than expected and feed back to client by returning a `DCWARN_PROPERTY_MODIFIED` warning.
- When reading:
  - If less data available than requested:
    - Return `DCERR_PROPERTY_SIZE_MATCH` and fail early (no transmission)
    - Return partial buffer and communicate effective bytes read in the `len` field.
  - If more data available than requested (some clients may not know the buffer size a priori)
    - Fail completely, if server wants all data be picked up at once
    - Only return number of requested bytes (e.g. file stream) and possibly buffer remaining bytes
    - Return `DCERR_PROPERTY_SIZE_MATCH` and return current buffer size in the `len` field.

A python module, for example, has no knowledge about the expected buffer size. Therefore, a first call from the client is made to the dcDevice_GetProperty() function with a zero buffer size. A proper buffer handler supporting a python client should always cover this scenario and report back the size according to the method listed above.
As you can see, there are some tricky aspects with respect to buffer handling. Improvements may be found in future, please check for deeper details in the API documentation [apiref] Appendix B.1.

3

Client variants

3.1 netpp master program

The netpp master control utility is very simple to use. When you call it without arguments, it will output the following:

```
Usage: netpp [<target>] [<property>] [<value>]
       <target>   : <hub>:<port> e.g. TCP:127.0.0.1:2008
       <property> : a property name
       <value>    : a value. The format of the value must
                    match its type, see below.

Depending on the number of arguments passed, this test program
has the following functionality:
[0] Show usage and list available hubs/ports
[1] Show property list of specified target
[2] Get value of specified property
[3] Set property to specified value

Available interfaces/hubs:
   Child: [80000000] 'TCP'
      Child: [80010000] '192.168.1.2:2008'
   Child: [80000001] 'UDP'
      Child: [80020000] '192.168.1.2:2008'
```

On the bottom you see a list of available Hubs or interfaces. These may have Port children (which stand for a successfully probed device). A typical session would now probably continue with an attempt to talk to one of these devices. We prefer to use TCP, because of it's session nature:

```
netpp TCP:192.168.1.2:2008
```

What we would get as answer:

```
Protocol Version 1
Checksum: 0426
Properties of Device 'DerivedDevice' associated with Hub 'TCP':
Child: [10000001] 'Demo'
Child: [10000002] 'Extra'
Child: [10000003] 'Test2'
Child: [00000002] 'ControlReg'
Child: [00000003] 'ControlRegH'
Child: [00000004] 'ControlRegL'
...[some lines removed]
```

This is the top level device list of the device properties. You can now query a single property via:

```
netpp TCP:192.168.1.2:2008 Test2
```

and get

```
Type : Integer [RW.]
Value: 40
```

Some properties have attributes, thus display children. You can query their attribute like you're used in various programming languages by appending a dot '.' and the name, like

```
netpp TCP:192.168.1.2:2008 Mode.Slow
```

This way, you can browse the property hierarchy of any netpp-speaking device.

3.2 The C/C++ API

The C/C++ API is fully documented in the API reference and is created from the source using the excellent documentation tool Doxygen. If you wish to generate the most up to date documentation, you will have to install the doxygen tool and run it inside the netpp top level directory. An online link to the API documentation can be found via http://www.section5.ch/netpp.

3.3 Python scripting

The netpp API can be accessed through python via an extension module. In fact, there are two modules:

1. `device.so`: The low level device access module
2. `netpp.py`: The mid level python'ish approach to property access

Normally, you would use the netpp module from your application by a simple import statement as shown in the example below:

```
import netpp

device = netpp.connect("TCP:motorserver")  # Connect to remote motor controller

r = device.sync() # Synchronize with property list and obtain root node
status = r.Status.Ready.get() # Query 'Status.Ready' property
if status:
    r.Motor.Speed.set(20) # Set motor speed
    r.Motor.Enable.set(1) # Turn on motor
else:
    print "Motor controller not ready"
```

The entire property hierarchy is encoded in the node class, the top node being the root node `r`. An internal checksum is built over a property list which is queried by the netpp module. When opening a connection using the connect method, the netpp module checks a local cache whether the device is known by comparing the checksums. If it is not found or when it has changed (differing checksum), the entire property tree will be queried from the device. Depending on the number of properties, this can take a while. It is open for discussion, whether a standard naming based approach within netpp should be defined for faster download of full property hierarchies, for example downloading an XML file via a stream style property.

3.4 GUI integration

Specific for the wxWidget environment, there is an extended XML framework, that allows to graphically design a dialog box or editor panel, insert widgets and directly map them to Properties without extra programming efforts. Depending on the memory capabilities of the embedded device, the XML dialog resource to configure the device can be stored in a property on the device itself. This allows a generic GUI control tool for any netpp capable device to use the following initialization procedure and fallback scenarios:

1. Initiate netpp protocol
2. Query `XRCFile` Property. If it exists, download the XML file and use the resource with the same name as the name given to the device node (root node)
3. If it does not exist, query `XMLResource` property and look for according local file. If not found, retrieve the file from the `XMLVendorURL`.
4. If that also fails, query the Property list from the device and build a Property Editor Table from the Properties found

The GUI wrapper framework is not part of the free/OpenSource netpp distribution.

The GUI integration is fairly complex and often very custom specific, therefore the GUI integration is offered as a separate professional service only.

4

Application notes

This chapter lists a few often practised applications for netpp or the DClib subset. Also, a few advanced techniques are described. If you are confused by a term or a specific filename that is not explained here, have a look at the `devices/example/` folder, in particular the Makefile.

If you are completely lost, post your problem in the forum (http://www.section5.ch/forum) or use the standard support channels

4.1 Prototyping and testing direct register access

Assume, you have created one or several registermaps for a I/O address space, for example for FPGA and CPLD hardware attached via different interfaces. Also, you have sorted out all the low level device access via device_read()/device_write(). Now you'd want to play with registers to prototype functionality that is subject to change. In this case, it makes sense to export all register definitions just as they would be properties. This is best done using the regwrap.xsl style sheet. It generates a group and properties with same names as the register IDs. The command to generate a register property XML file is:

```
xsltproc -o register_properties.xml $(NETPP)/xml/regwrap.xsl device_description.xml
```

To automate this process, it is best to use a Makefile rule. A 'make' should then cover up the entire development flow. You then use the **generated** (register_properties.xml) as actual `DEVICEFILE` to pass on to the `prophandler.mk` auxiliary rule file.

4.2 Creating API/Hardware documentation using XML/Docbook

The procedure to create a basic hardware reference to your register and property descriptions is already implemented as rule in `xml/prophandler.mk`.
To create the documentation for the standard example, enter `devices/example` and run: **make device_properties.xml**
Nodes from this documentation can again be included easily using the `<xi:include>` statements into your DocBook based documentation. As this documentation is written in in DocBook using the XML editor described above, this is simply included via selection of the desired documentation node in the generated document via the "Copy as Reference" menu and pasted into this documentation.
An example of an included address map documentation is shown in Table 4.1. A more detailed example of a full device documentation is shown in Section A.1.

Offset [Span]	Name(Id)	Access	Description
0x00 [1]	Control0	RW	Control register 0
0x00 [2]	Control	RW	Concatenated Control0 and Control1 register (16 bit width)
0x01 [1]	Control1	RW	Control register 1
0x02	LED_Slot1	RW	LED slot 1
0x04	LED_Slot2	RW	LED slot 2
0x06	LED_Status	RW	Status register. Writing a 1 to this register clears the status bit and corresponding LED.

Table 4.1: Address map FPGA_i2c starting at MMR base 0x00000000

4.3 Inclusion and derival of device descriptions

4.3.1 The <xi:include> statement

Assume you have a family of devices that are all making use of the same data acquisition device or sensor. As a C programmer, you are used to a header that you include, plus you link to a library so that you don't have to spell out things again or be condemned to a lot of copy-pasting, when a new revision of the sensor comes out. Likewise, you would wish for an XML device description, so that your entire family of devices can just include one single source definition.
This is achieved using the `<xi:include>` directive. This pseudo node is very powerful, you can include specific nodes from another file into your device description.
For example, if you were just 'linking' to a register map of another device, you would use a statement like

```
<xi:include href="sensor_device-1.0.xml" xpointer="reg_i2c" />
```

The `href` attribute is a direct reference to the source file, the `xpointer` refers to the ID of the XML entity, in this case a `<registermap>` node.
If you want to export a hardware definition to another file, you have to provide it with an `id` attribute.

4.3.2 Derival of a base class

Assume you have already created a complex hardware definition for a device, plus the software handling around it. Your device may already even be out in the market. Now you are designing a better version of the device, which has some extra features but is else downward compatible to the 'old' model. But you do not want to keep several software versions around for various devices, it would be nice, if one single software could serve them all. This can be achieved using derivation: A base device class is defined, and another class can inherit its properties - like in C++ or Python.
The `example.xml` device description in `devices/example` of the source distribution demonstrates how a derived device inherits the properties of a base class, can override properties and implement special features.
Note that unlike in C++, device properties are 'runtime structures', thus a multi-derived class hierarchy should be carefully designed. Generally, it is a good idea - like in C++ - not to overdo derivations but keep things simple and always evaluate the inclusion alternative described in the previous section.

Technically, when defining a set of classes, they will end up in one file (or make use of inclusion from other files). Note that when building a device control library, all device nodes in the specified device description file will be translated. So you will end up with a software, that has knowledge about the features of the entire device family.
The question is still left, how the target firmware exactly reports to the connecting client, what device type or class it actually is. This is described in detail in the next section.

4.3.3 Advanced derival using dynamic properties

Some devices may be more complex such that they allow properties to grow at run time. As described in the previous section, class derival is a mechanism to augment functionality of an existing description without having to entirely reimplement it. Using dynamic properties as mentioned in Section 2.6, a device can have a basic set of default properties plus add other dynamic properties on the fly.

4.4 Multiple device 'driver'

If you have a device file with multiple device definitions, your target software will have to select one. You could see it like an USB configuration, depending on startup conditions, the device selects a configuration from a few possible ones and reports it to the client.
The dcDevice_GetRoot() function on the client side obtains the device root node in the beginning of a new communication. On the device (server) side, this calls the function `local_getroot()` to obtain the current device's root TOKEN. Thus, the index of the selected device configuration is reported to the client by the return value of the `local_getroot()` remote procedure call. Again, see the example in `devices/example` how this is realized.

4.5 Device proxies

In some cases, it might be necessary to tunnel connections to remote devices to another protocol. For example, when accessing a group of wireless sensor devices using a different protocol, or when it is required to run netpp through an encrypted static peer connection, a proxy might be desired which forwards configuration requests to a different logical network or in netpp terms: Hub.
This is a situation where a server needs to be master and slave at the same time. It may:

- Act as a slave towards the front end protocol (e.g. TCP)
- Act as a master towards the back end protocol (e.g. USB)
- Possibly handle out of band messages from the back end protocol and take local notion
- Possibly act as a master towards the front end protocol again to notify clients of important changes (alerts)

Whenever a back end protocol (be it netpp or a different low level protocol) is to be forwarded to a netpp server, the proxy code comes into place.
Proxies are like a normal netpp device, but with a basic set of parameters, which must include the basic properties listed in Table 4.2. A client side proxy Hub calls a specific secondary probe sequence when it has found a proxy Hub on the network through the discovery protocol. This sequence is consisting of a few remote procedure calls to the proxy itself, more precisely, a local probe request is made to the proxy which again discovers attached devices and reports them back through a simple iteration through the basic properties.
Typically, the devices behind the proxy are called **Endpoints**. This somewhat corresponds to the endpoint notion of a USB device, however in this case, endpoints can be netpp-aware remote

devices again. In theory, a device behind a proxy does not necessarily have to be an endpoint and can again be a proxy, but this is beyond the current scope of functionality.
A proxy hub has the Hub name "PRX". A netpp master, if compiled with proxy support, will poll for Proxies as well and list them like a TCP or UDP hub. If the proxy detects Endpoint children on the basis of its tunneling protocol, it will list them and allow them to be accessed like a normal remote device.

Property	Type	Function
Scan	COMMAND	Issue a device scan on the proxy
Peers	ARRAY of STRING	Endpoint device ID found during scan. Obtain the number of devices found by querying the .Size member.
Open	INT	Open device with given index
Open	STRING	Open device with explicit ID
Close	INT	Close device with given index

Table 4.2: Basic proxy properties

For further details of proxy server development, please refer to the example source in `$(NETPP)/devices/proxy`.

4.6 FPGA/ASIC register map definitions

Even if you are not thinking about networked communication yet, you could just make use of the DClib XML framework to describe and document the register maps of your newly designed sensor, ASIC, or FPGA solution. The reasons for this are mainly that you will avoid some typical 'desynchronization problems' during development, i.e. when your colleague changes the register map, but has forgotten to tell you, let aside the people writing the hardware reference manual.
So, a possible design organization might look like:

1. Keep the entire device description in one place - the device XML file
2. Maintain this device XML file using revision control software (preferrably Subversion)
3. Set up your toolchain that all code, VHDL packages and documentation parts are re-generated when a new version of the XML file is checked out from the revision control server.

Sooner or later you might run into limitations with the current device description dialect, for example because you need to introduce new XML nodes that describe specific properties of your hardware. For example, netpp is good at describing register maps, but it does not have a language yet for command based interfaces. However, a command style structure can always be emulated with Property based RPC calls. For extending netpp, see Chapter 6.

All hardware relevant topics are now covered by a separate SoC documentation [soc] Appendix B.1.

4.7 Synchronization and Versioning tricks

If you are a Subversion user, you can insert Revision ID strings into STRING type Properties, provided that your device description file is revision controlled. The details are found in the

Subversion documentation, but basically, you insert a Subversion keyword tag `$Rev:  $` into the Property string. When the file is commited, the current revision will be inserted. However, you need to prepare Subversion to recognize these key words:

```
svn propedit svn:keywords device.xml # replace device.xml by your device description
```

Then make sure that the keyword 'Rev' is listed in the first and only line.
After the next commit and another build, the current revision will be inserted in the revision tag in your device file. When developing in a team using Subversion, this allows quick synchronization checks between documentation, software and firmware.

5 XSL converter reference

All XSL converters take certain parameters to control the output or just select a particular node from the device description. Depending on the style sheet class, there are some common options, some are specific to the style sheet.
Note: All SoC and VHDL specific XSLs are documented in the SoC reference [soc] Appendix B.1.

5.1 Style sheets for C source code conversion

Common options to source code conversion XSLs:

srcfile
: [string] The source file used to create the output from (filename of device description XML). This is typically passed from a Makefile.

selectDevice
: [string] Either a string containing the id of the desired device or [integer]: an index (1..n) matching the position in the description file.

regprefix
: [string] When specified, use as prefix to the C header register definition. By default, this is 'Reg_'.

useMapPrefix
: [boolean] When 1, use the parenting register map name as prefix for output symbol generation.

5.1.1 registermap.xsl

This XSL creates the C header register definitions and address offsets for inclusion by the software driver.
Specific parameters:

convertBitfields
: [boolean] When 1, also convert bit field definitions into the resulting header file

Example snippet of created register.h:

```
#define FPGA_i2c_Offset REGISTERMAP_OFFSET(0x0000)

#define Reg_Control0                                    (FPGA_i2c_Offset + 0x00)
#define Reg_Control1                                    (FPGA_i2c_Offset + 0x01)
#define Reg_Control                                     (FPGA_i2c_Offset + 0x00)
```

5.2 Style sheets for graphical output

These convertor sheets are mainly used to generate meaningful graphics displaying detailed register information (such as accessibility of single bit fields, etc.).

5.2.1 reg2svg.xsl

This XSL outputs a specific register configuration to a SVG file.
Parameters:

register
: [string] Mandatory. Specifies the register to be converted into the graphical representation.

useMapPrefix
: [boolean] Prepend parenting register map name to register name output

For an output example, see Fig. A.1.

5.2.2 regmap2latex.xsl

Creates a full register map reference using LaTeX and other optional packages. This file is part of the professional support package.

5.3 Auxiliaries

5.3.1 linkerscript.xsl

This XSL creates a linker script from the `<memorymap>` node, representing the correct allocation addresses for the bare metal physical address location of the firmware, typically the boot ROM. For now, this is undocumented.

5.3.2 gdbscript.xsl

Creates a gdb script containing several function definitions for interactive access of memory mapped registers, including verbose bit dumps, etc. Undocumented.

6

Extending netpp

The netpp protocol and property design is kept somewhat atomic in order to stay robust. However, it is not always the most efficient solution for specific implementations. It has turned out over all those years of practical usage, that all extended functionality can be either implemented 'on top' using Properties or on the physical/logical transport layer. Still, many things are open for discussion as there are numerous aspects from different use cases.
This chapter lists a few possible directions concerning extensions.

6.1 Upward/Downward compatibility considerations

- The netpp protocol
- Token and Property names
- The device description XML language
- The device description itself

These are the important points to keep in mind to keep compatibility:

The netpp procotol

There's one rule: Protocol changes must be authorized, and the protocol number must be increased accordingly. The protocol must be downward compatible, i.e. newer client versions must be able to speak to old servers.

Tokens vs. Names

- The TOKEN and encoding can change (except reserved TOKEN values)
- The name should not! If it does, see below.

The device description language

The language may change and be subject to enhancements. Changes in the language will increase the version number. For each change, a translation style sheet is required to convert old device descriptions into the current language version.

The device description itself

You, as the developer or device vendor are responsible yourself for the versioning and identification. It is recommended to use the `Vendor` top level struct Property (see Table 2.4) and populate with your required identification tags.
Typically, a GUI application (front end) to remote control an embedded device would apply the following strategies to fully sync with the controlled device:

- The device has a GUI description stored, which is retrieved by the front end and used to build the user interface (such as in the netpp wxWidgets interface Section 3.4)

- The front end has some knowledge about the device class and assumes a few standardized properties. This set of standardized functions may grow, so the situation may occur, that a new front end speaks to an old device. If a recently standardized Property does not exist on the target, a semi-userfriendly method is, to disable the concerning widget and warn the user via a status log window.

6.2 Style sheets

The current set of style sheets included in the netpp distribution mostly cover code generation for C source and headers, plus some auxiliary translators to create documentation templates. However, there are many more possibilities to translate into other formats by writing another style sheet. The XSLT translation language (which is again XML) is best covered by the W3C resources [w3c] Appendix B.1.

6.3 Property Naming

By default, netpp does not tell you how to name Properties. Apart from using special characters, there are no restrictions. However when you create an entire similar class of devices, it will make sense to stick to a consistent property naming. For example, when you wish to emulate various logical layers of device class implementations like HID devices, Cameras, etc, you will have to define a fixed set of properties.
For that purpose, there will be a read-only `DeviceClass` property in the netpp 1.0 specification. See also Table 2.4.

6.4 Device Class design roadmap

netpp is far from defining a unified standard. It is therefore open to all kind of enhancements and further top level standardizations. The short summary how a roadmap to a better standard could look like:

1. Agree on the fact that a new device class standard makes sense
2. Define registered device class name
3. Define minimum/atomic functionality set for this class (Minimal Required Property List)

The goal is, to keep a standardization process:

- As open as possible
- As unbureaucratic as possible (smart web registration procedures instead of heavy standardization commitees)
- As automatized as possible (Web services)
- As democratic as possible

Bottomline: Look at XML and other open standards and follow the scheme that has worked out before.

6.5 TODO

Here is a list of things that are missing or known as undefined in netpp 0.3:

- No local interface configuration supported in netpp 0.3/procotol v1.0. A Hub has a fixed packet size.
- Only 32 bit signed integers supported. All other bit specific data structures are handled only by buffer types or via meta properties (structs)
- There is no specific definition for character encoding. All enhancements are to be discussed on top of netpp/DClib based on a descriptive property set.
- A dictionary type. This is a dynamic property node which might also allow creation of new node members, depending on implementation.
- netpp has no built-in method of communicating device specific error descriptions. These will be communicated using the standardized method using the optional ReturnCodes dictionary type (dynamic property) node.

A

Examples

A.1 A generated example device documentation

Fig. A.1 shows a generated (via `reg2svg.xsl`) register structure schematic.
The following section is a referenced (not copied) documentation entry for the derived device from the standard example.

A.1.1 DerivedDevice

Device Revision: 0.1
A derived device, basic on ExampleDevice. It includes extra features, and overrides a base feature.

Properties

Property	Type	Flags	Description
`Extra`	INT	RW	This is an extra property, adding to the base class.
`Password`	STRING	RW	A hidden password string
`Test2`	INT	RW	This is an example of a property overriding the previous definition in the base class. See base class.
`DynLimits`	INT	RW	Demonstrates dynamic limits (that may depend on other properties)
`Demo`	INT	RW	A demo configuration variable
`GPIO`	ARRAY	RW	This is an reference implementation for a GPIO bank with multiplexed pin functionality

Table A.1: Extra Properties

Table A.2: Things that do not work in netpp (but in devdesc)

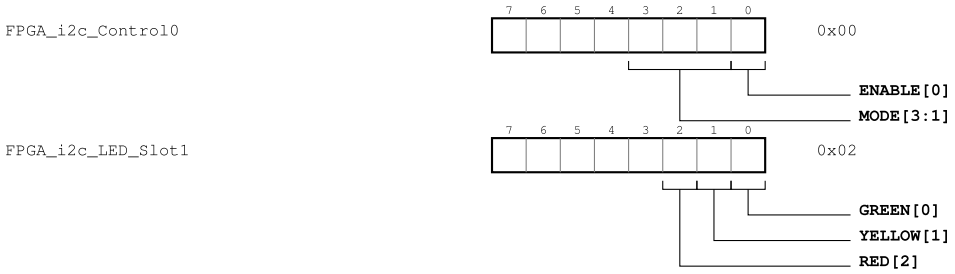

Figure A.1: Example of generated register structure

Property	Type	Flags	Description
Container	STRUCT	RW	

Table A.2: Devdesc_Extra_Tests

A.1.2 ExampleDevice registers

'FPGA_i2c' core registers

Bit(s)	Name	Description
3:1	MODE	Operation mode: 0: slow, 1: fast, all set: idle
0	ENABLE	1: Enable device, 0: Standby

Table A.3: Control0 register Address: 0x00000000

Bit(s)	Name	Description
15:8	HI	Higher half of the register
7:0	LO	Lower half of the register

Table A.4: Control register Address: 0x00000000

Bit(s)	Name	Description
2	RED	Red LED switch
1	YELLOW	Yellow LED switch
0	GREEN	1: Turn on green LED, 0: off

Table A.5: LED_Slot1 register Address: 0x00000002

Bit(s)	Name	Description
5:4	RED	
3:2	YELLOW	
1:0	GREEN	

Table A.6: LED_Slot2 register Address: 0x00000004

Bit(s)	Name	Description
3:2	BLUE	
1:0	GREEN	

Table A.7: LED_Status register Address: 0x00000006

A.2 Rapid design of a System on Chip

Analogous to various system builder software tools, I/O maps can be generated on the fly using devdesc device descriptions. However, there is a huge number of design options and ways of implementation such that it is almost impossible to define a generic standard. For example, a existing system may restrict you to the usable bus width and addressing modes whereas the attached I/O device may have different properties. In the following sections we will deal with a few different scenarios.

A.2.1 32 Bit CPU peripheral interfacing

A typical 32 bit CPU has a 32 bit address bus and a 32 bit wide data bus. In a FPGA soft CPU design, access to peripheral controllers is typically implemented using memory mapped registers (MMR) in a specific I/O addressing section. To save extra addressing logic, these MMR registers map on a 4 byte boundary, i.e. the two least significant bits in the address are zero. Now this can turn into a little problem, when we have to map an existing peripheral map that has a different data width into this MMR space.
To define the address mapping to various bus widths, a dummy register map with the name *MMR* is created in the device description XML file. This map can contain one or more pseudo register definitions that again contain bit fields specifying address decoding portions, the 'MMR config bitfield'. These have a internally standardized (with respect to the conversion style sheet) prefix: `MMR_CFG_`. For each <registermap> node specified in the device description, a corresponding MMR_CFG_**<id>** bitfield must exist. For example, you define a register map with id 'i2c_controller', then you have to define a bitfield called `MMR_CFG_i2c_controller` in the dummy register of the MMR registermap.

It is mandatory to set the `id` attribute of the register map for the conversion to HDL to operate correctly.

The MMR config bitfield is used to slice a local address portion out of the MMR address space. A peripheral map such as an i2c controller typically gets by with eight registers, therefore only three address lines are needed. In case of a 32 bit wide addressing however, we do not slice the three least significant bits but skip the first two LSBs. So our `MMR_CFG_i2c_controller` bitfield definition would look like:

```
<bitfield lsb="2" msb="4" name="MMR_CFG_i2c_controller">
   <info>Configuration address range for i2c unit with 8 bit addressing</info>
</bitfield>
```

During the conversion process into VHDL, this definition creates a subtype:

```
subtype     BV_MMR_CFG_i2c_controller is integer range 4 downto 2;
```

This subtype is then used to slice the required address bits from the MMR bus address in the generated VHDL code.

A.2.2 Automatic updating

Once you have created a registermap inside a device description, the automated procedures typically create:

1. A hardware decoding logic in HDL
2. A C header file

3. Documentation

Within a SoC design, it is desired to control this all with one call to **make** to keep everything in sync. So, whenever a register definition is changed, the following happens under the hood when executing make:

1. The C program is being recompiled with the new register addresses
2. The ROM generator creates a memory initialization file in HDL
3. The simulation or netlist file is recompiled

To simplify the building and synchronization of all files, several tools or files are involved.

soc/busgen.mk
: Contains the rules to create the decoder instances and the system map package from the device description $(DEVICEFILE)

soc/core/buildrom.py
: Python script to build a VHDL instance of a preinitialized RAM from an ELF format executable (supports ZPU and MIPS architectures only)

These tools are typically found in the distribution of your SoC evaluation distribution.

A.2.3 Multiple instancing of peripheral controllers

Assume a controller instance mapped somewhere in the MMR space. Now if several instances are desired, for example a second UART, another portion of the MMR address might be needed to select the corresponding peripheral unit. This has to be encoded in two domains:

1. The HDL domain
2. The peripheral driver C source code

A simple way to cover the C source side would be to duplicate the register map and use different offsets. It is more elegant however, to define a macro that calculates an address offset for each device according to the device index. This device index is sliced from the MMR address on the hardware side, like with the MMR config bitfield from above. The selection of the device index portion from the MMR address is therefore defined with a `MMR_UNIT_<name>` bitfield entry in the MMR pseudo map. The resulting define `BV_MMR_UNIT_<name>` in the VHDL code can then be used likewise. On some conversion systems, the entire instancing of peripherals is controlled and generated automatically by a specific property section inside the device file.

One might be tempted to use the two unused LSBs for the device index. However, this is not a portable solution and very much depends on the CPU allowing 32 bit wide data addressing to addresses that are not on a four byte boundary.

For example, a macro to access a specific UART device register map:

```
#define uart_dev_mmr(dev, x) \
        ((volatile unsigned long *) x)[(dev << (MMR_UNIT_uart_SHFT-2))]
```

Care must be taken with using the MMR_UNIT_<foo>_SHFT value, depending on the address width, this must be decremented by 1 for 16 bit, 2 for 32 bit (as for the case above).
The UART register is then simply accessed like

```
uart_dev_mmr(1, Reg_UART_THR) = 0x55;
```

This approach only works when sufficient address space is reserved between different controller MMR maps. Otherwise, the address decoding must be done manually with specific functions.

B

References

B.1 Bibliography

A list of documents and further pointers:

[w3c] **The W3C website**
URL: http://www.w3.org/

[xxe] Pixware
The XMLmind XML editor
URL: http://www.xmlmind.com/xmleditor/

[netppres] **The netpp resource page**
09/2011, section5::ms <hackfin@section5.ch>
URL: http://section5.ch/netppres

[apiref] **The netpp API online reference**
09/2009, section5::ms <hackfin@section5.ch>
URL: http://www.section5.ch/doc/netpp/html/

[devdesc] **The device description XML dialect**
04/2005, section5::ms <hackfin@section5.ch>
A set of schema description files, part of the netpp distribution
URL: http://section5.ch/netpp/

[soc] **The gensoc package**
section5::ms <hackfin@section5.ch>
The gensoc package is a netpp-addon for generation of System On Chip register map decoders for VHDL based bus interfaces (Wishbone or local bus).