netpp/dclib – A universal library for embedded device remote control

Martin Strubel

April 26, 2011

Revision:

`v1.0 $Rev: 303 $`

Abstract

A standard task when designing an embedded device is, to choose or design a remote protocol for communication with a client and a user interface. Moreover, embedded devices mostly use registers (or local variables) for configuration. Wrapping these low level hardware entities into abstract properties that may eventually be altered via a user interface, is a recurring and tedious procedure that every developer would like to see automatized. The **netpp** (an acronym for *Network Property Protocol*) library respectively its **dclib** (*Device Control Library*) portion is a set of tools and source code framework that simplifies the design process of remote control and user interfaces (even down to hardware development) greatly by:

- Allowing a hardware developer, programmer, or UI designer to write a *single* XML file containing a hardware and a "property" description
- Generating register maps and arbitrary source code (normally C) from that description by a simple "make" procedure, eventually compiling it into a binary for execution on the device
- Remote control of register parameters or C variables via various interfaces (TCP/IP, UDP/IP, USB, ...) by scripting, simple command line clients, graphical user interfaces or custom applications.
- Allowing to build a simple user interface for graphical, data type sensitive remote control from the property information provided by the device protocol

Name	Description
dclib	Device control library. Includes the entire basic framework for source code generation from XML. A device control library within this context does not necessarily have to be **netpp**, other concepts can be derived from the dclib framework.
netpp	Network Property Protocol. Special case of the dclib for remote control which is now considered "default". Includes a simple peer to peer protocol which works on character or packet based interface. The netpp base library is implemented in C.
devdesc	Device description XML language dialect.

Table 1: Library component naming

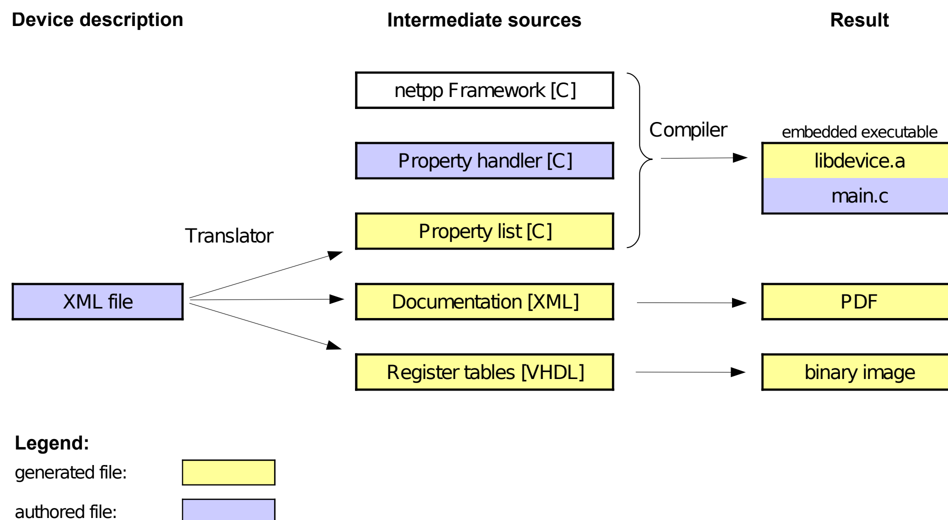

Figure 1: Development flow schematic

1 Introduction

The early dclib framework has been designed in 2003, was rewritten with the network functionality (as netpp) in 2005 and is now (2010) considered mature and universal enough for public release. To elaborate on the naming scheme, see Table 1.

Fig. 1 demonstrates the design flow of a device backend implementation. The files that have to be written by an author are marked blue, generated files yellow.

The library is designed very light weight, such that it runs on a simple MCU or even FPGA, with one requirement: The C compiler must be able to handle 32 bit integers. On the front end side, the device independent part of the library allows to talk to all netpp powered devices. It has proven to be more efficient and much simpler than XMLRPC ([xmlrpc] Appendix B), while providing similar functionality. Also, it is supposed to have a better thread safety. Apart from being cross platform compatible by design, it has API interfaces to C++ and to the Python scripting language, which allows automated functionality tests.

2 Property concept

A hardware unit such as a system-on-chip device normally has a command interface such as I^2C and an internal register map. A more complex MCU may have its configureable, global options organized in records, more precisely, variable structs located at a fixed memory address. Whether an option is controlled through a bit, a register value or a variable in memory, is not exposed to the user. All these entities are mapped into abstract *Properties* by the *devdesc* description language. For example, for a status bit in a register, a register and bitfield declaration is created. The according *Property* is another declaration, referring to the previously declared register by its name identifier.
Assigned to a Property is a type field describing the property data type, e.g. INTEGER, or COMMAND. The identifier for a property is a string, for example, "Reset". Properties also have flags, for example, READONLY, VOLATILE, etc.

2.1 Property resolving/addressing

To be able to query the capabilities of a device, the properties are organized in a tree structure. The device itself is represented by a root node, its children can be top level properties or struct type properties that serve as namespace container to group related options, e.g. `Window.X`, `Window.Y`.
The netpp function set allows to walk and iterate through the tree structure and thus query the entire property table of a device. Depending on its data type, a Property can have other Properties as children (e.g. minimum/maximum allowed values or a choice of several valid operation modes). This allows to build a minimal fallback user interface for device control from the embedded property description.
The actual addressing of a Property is not directly done via its string identifier, but its *TOKEN* (which is a unique 32 bit property ID). To elaborate, we look at the standard sequence of a front end side property control:

1. Retrieve the TOKEN of a device property by its name via the `ParseName()` function
2. Initialize a *Value* proxy structure for setting or retrieval of a property value.
3. Call `SetProperty()` or `GetProperty()`, passing the TOKEN and the pointer to the Value proxy.

The property concept requires a little different strategy from a functionality design point of view. Normally, a hardware designer would layout a register map, define the hardware behaviour on a low level, then pass his circuit on to the software developer who typically has to dive into bit fields and interface them to a front end user such that all illegal settings that could make the hardware crash, are caught. In the netpp case, one is encouraged to a *top to bottom design* rather than the other way round: *Think about the properties first*, then map them into registers or even bit and mode fields within a register.
The property hierarchy allows enhanced types such as the above mentioned structs, or arrays, arrays of structs, etc. The latter allows to map complex register table sets that cover up parameter sets for different operation modes (contexts) into property structure arrays without overhead. See example in Section 6.1.1.

2.2 Compatibilities and enhancements

The XML device description language and device implementation are independent from the protocol, thus, networking functionality of the hardware is a priori not a requirement. The only points to keep in mind are:

1. The XML dialect can change without implications to the backward compatibility
2. The TOKEN values can (internally) represent a pointer address on the client, or an index, or an opcode. For the front end, it is just a 32 bit value and is never interpreted directly.
3. TOKEN values for properties can change, so they must never be referenced directly. However, they can be cached and validated using a checksum both on the device (backend) and client (frontend).

2.3 Special features

The XML device description allows to specify certain primitive validation elements, like minimum and maximum allowed values for INTEGER type properties. These limits are spelled out as attributes of a property and are instanced as children of the concerning property: 'Min' and 'Max'. Again, such a limit instance is a Property with a static value.
If a validation is more complicated, it is normally carried out on the device itself using a handler procedure. If the handler receives an invalid value, it will either adjust the value and return a warning code (meaning, the transaction was completed under special notice), or refuse the transaction by returning an error code.
The effects of a property alteration can be taken on further: For example, a setting of a Property could affect the state of another Property. Best example: A full "Reset" command triggered from the controlling client (master) requires all properties to be reread. This case will return a warning code "All properties have changed" to the client, however, it might be desirable to only notify *specific* property widgets. This is done via EVENT type children attached to a property. Thus, you can implement the functionality for a user to change or override a configuration and notify him or her of the consequences to other configuration settings.

3 XML device description – devdesc

The XML tag language with its well defined validation framework seemed to be the best base for a device description language. Moreover, translation into other formats can be handled well using XSLT stylesheets, which enables a light weight solution that does not require a parser. Again, the XPath language to extract XML node elements is a well defined standard. The details of this proposed standard are covered in [devdesc] Appendix B.

3.1 Authoring/Validation

For creating the device description, it is highly recommended to use an XML editor that can process schema language extensions, in particular the XSD format. For dclib/devdesc, the preferred editor is

the XMLmind XML Editor ([xxe] Appendix B), see also screen shot Fig. 2. Customized style sheets are also provided for this editor, such that the editing of device properties is fairly easy. The author can fully focus on the functional design of the property set, all formatting and validation is taken care of by the editor stylesheet.

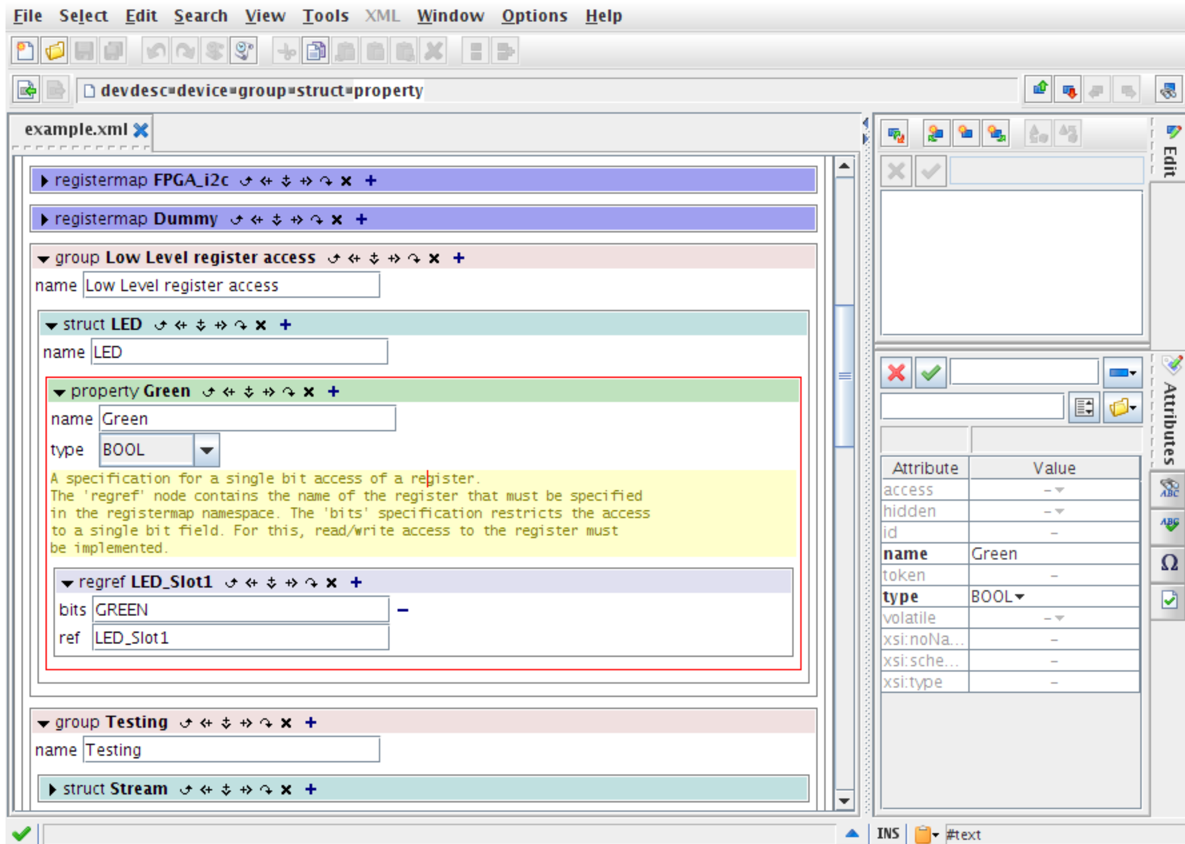

Figure 2: XMLmind XML Editor

3.2 Translation using XSLT

To translate the XML source into the various intermediate formats for generation of the software library, register documentation or even VDHL register maps (for programmable logic chips or ASIC designs), a set of XSLT style files is provided. Each XSLT translator sheet translates the XML structure (which must accord to the Schema language description) into the destination format. Using common Makefiles (processed by GNUs **make** tool) to define the translation rules, a simple 'make' call handles the entire translation procedure up to the binary compilation of the embedded-side software and keeps changes in the XML device description source in synchronization with the end result.

4 netpp inside – The protocol

4.1 Master and Slave

Before we examine the protocol philosophies, we should agree on the master and slave terminology. Obviously, a protocol turns out to be most simple if there is a dedicated master initiating a protocol

transaction and the slave responding to it via an acknowledge message. Thus, a slave never initiates a connection. The disadvantage of this simplicity is, that we have no proper notion of an interrupt or attention message. However, we will see later, that, using network functionality, we can symmetrically reverse the roles, i.e. a slave can simultaneously be a master.
The dclib portion (merely, the property handling and the XML description) is protocol independent. Thus, a dclib featured library can be compiled for a master, that accesses a simple device via a custom, preferrably register based protocol, for example I^2C. However, this implies that this library is device specific and needs to be updated for extended device support.
The netpp philosophy targets at pushing this master intelligence down to the device to be remote controlled: since most devices nowadays have at least USB and/or Ethernet and 32 bit capable controllers with reasonable amounts of memory, the netpp protocol introduces another abstraction layer between master and slave, such that a master can talk to all netpp capable devices, independent of CPU architecture, low level protocols or local variable structures.

4.2 RPC – remote procedure calls

In short, we list some of the important netpp properties and features:

- Works packet based, i.e. on any streaming or packet based interface
- Adds a 14 byte header per transaction
- Splits large streams into packet fragments
- Uses a simple atomic set of RPC (remote procedure call) primitives to tunnel API functionality between master and slave

The simple set of RPC packets used in netpp features all functionality described in Section 2.1. The basic RPC set is shown again in Table 2.

RPC call	**Description**
`Status()`	Status query and response
`Select()`	Select property node in hierarchy
`ParseName()`	Resolve Name into token (within parent namespace)
`GetName()`	Get Name of token
`Get()` / `Set()`	Get / Set value of property
`GetProto()`	Get property prototype (Type, attributes, ...)

Table 2: netpp RPC primitives

The protocol handler tries to keep as much symmetry between master and slave as possible, such that the common source code usage among master and slave is optimal. Using a network protocol, a slave can thus be a master as well – with minor modifications to the main program.

4.3 Multi device/interface capabilities

As we have learned, a master no longer needs to have device specific libraries lying around, when netpp capable. However, it must have a notion of the interfaces that netpp is based on (such as TCP/IP, UDP/IP, USB, etc.). In case of USB, the transport layer is also heavily dependent on the USB slave controller chip, so there may be several implementations. To support this interface diversity in a flexible way, the abstraction of a so called *Hub* is introduced.
When the master application is started, the supported Hubs can be queried from a local tree structure – which is in fact nothing else than another (local) Property description, although it is created dynamically. This dynamic property structure also supports a probe functionality to look for attached devices, which represent as a *Port* property node within the Hub structure. Upon successful opening of a Port, a device object instance is created and a handle (Pointer) is returned to the user. All RPC operations mentioned above are performed on that device object handle.
On network interfaces (TCP, UDP), a simple discovery service is implemented in netpp using UDP broadcasts.

5 Device implementation

5.1 Device side program code

Not all register functionality can be wrapped automatically, also, the access to the registers needs to be implemented by a `device_write()` and `device_read()` function. This function must translate a linear register address into the appropriate I/O commands on the hardware interface side and is typically implemented in the C programming language. Several devices mapped into a multiplexed address space can be encoded using specific address offsets (32 bit) per defined register map.
For complex access, handler routines ("Getters and Setters") have to be implemented per property. This is mostly the case for data buffer transfers between devices. In this case, a handler needs to provide target or source address of the buffer and an UPDATE or RELEASE action for the buffer, once the transfer completed.
Then, the main loop must be taken care of. The device must consecutively listen to the netpp protocol and respond to property transaction requests. So, this listener function is normally called from the main loop. Special care must be taken to return early from protocol handlers such that the protocol does not time out, meaning, no blocking routines are allowed inside the protocol handler.
The other option is – provided that the underlying operating system of the device provides the functionality – to use multitasked threads. One thread can then deal with the netpp handling only, other processes (such as data acquisition) can run simultaneously. This functionality is completely up to the main program, however, when netpp is compiled unter a Linux OS, *pthread mutex* handling is automatically built in. It is thus possible to run the netpp protocol from within different threads of the same program, however, this approach is not recommended.
All programming details are documented in the netpp API documentation [netppapi] Appendix B.

5.2 Device classes, derivatives

The device description XML dialect allows, to derive functionality of a device class and create an enhanced or similar device with the same base functionality without having to copy the entire property list. This makes it very easy for a vendor to provide a platform that is hardware configureable and runs with *one* uniform software. So, maintaining an entire class of similar devices with different functionality is a matter of *one* XML source file.

6 Application notes

6.1 Reference implementations

A list of product proof examples for device remote control implementations using netpp:

6.1.1 Network camera with complex sensor properties

For a Linux based IP-camera prototype, it was required to use several complex system-on-chip (SoC) sensors, possibly in a multiplexed mode, to acquire images in different operating modes, depending on the environment situation. The sensors are normally controlled via I^2C (or Two Wire Interface) and have a quite extensive number of registers and properties, thus manual addressing of these properties within the program code was not an option. Nearly all I^2C register properties of all existing sensor head boards are wrapped into several sensor specific device files, which are included in a modular way into a main camera description file. The basic common functionality is defined by the camera device base class, for each pluggable sensor module, another device class is derived from the camera base class (see also Section 5.2). Depending on the plugged in camera module, the 'videoserver' determines, which derived device class is to be used, and communicates the according device root node to the front end user.

Further, some SoC sensor devices use different contexts, for example, exposure time in video mode and exposure time in snapshot mode. For both operation modes, a context area is reserved, containing shadow values for a raw sensor register set. Depending on the used mode, the entire context area is copied into the raw registers.

To address properties within these contexts, the user defines a struct array which is accessed by name as follows: `Context[0].ExposureTime`. Offset, size and base address of these context tables can be specified in the XML description.

6.1.2 Remote display for network camera

For the network camera described above, video transmission standards such as Motion JPEG (viewable via HTML browser) is a viable option to transfer a video stream, however in some cases, raw video data or meta information needs to be sent to a client. As a consequence, a display server for a client PC was implemented as a netpp device, such that an embedded camera can send images and meta information (like traced geometry data) to the display in various video modes (RGB, Bayer, raw data, etc.)

6.1.3 Remote controlled networked medical image scanner

The netpp library is designed for data buffer exchange with minimal copying overhead. Thus, it can be used for more complex streaming or file transfer. In the concrete implementation of this scanner, the following functionality was desired:

1. Complete remote control and configuration via customer software from several clients at the same time
2. Session management: Several users should be able to run a scan. If a user is claiming the scanner, the ID and location of the user is reported to others.
3. Non-interrupting (streaming) data transfer of scanner data over a reliable TCP protocol, or alternatively over USB for standalone operation.

6.2 Memory footprints

Table 3 shows the typical memory sizes of a simple netpp slave library (libfiletransfer.a) with up to approx. 10 properties, a few file I/O handler functions and TCP plus UDP/IP functionality.

Architecture	**No optimization**	**Optimization '-O'**
i86 (32 bit)	54 kB	47 kB
x86_64	56 kB	47 kB
i86 Windows (mingw32)	39 kB	30 kB
Blackfin	37 kB	31 kB
armv5tel	58 kB	41 kB
armv5tel (thumb)	42 kB	33 kB

Table 3: Memory footprints of libfiletransfer.a for various architectures

A Discussion

In the past few years, netpp/dclib has proven to be an efficient tool for rapid prototyping as well as suitable for industrial products. However, the implementation focuses on simplicity and light weight rather than extensive network safety or real time considerations. Integrating all various aspects on interfaces and different platforms will require support by a larger community of developers and users. Therefore, you, as reader, are encouraged to evaluate, implement, improve and feed back. This is considered the next step for netpp/dclib: Seeing, if it can be turned into a standard for a larger user base by going OpenSource.

A.1 Future plans

For the future development, the following items are on the list:

1. Figure out licensing details and naming
2. Set up an OpenSource repository for netpp
3. Implement device database and user / vendor community
4. Collect device data with help from community
5. Work towards an open, license free standard

Enhancing the device description language by schematic descriptions and footprint data is under scrutiny and subject to discussion.

B Bibliography

A list of documents and further pointers:

[xmlrpc] **The XMLRPC website**
URL: http://www.xmlrpc.com

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

[netppapi] **The netpp API documentation**
09/2009, section5::ms <hackfin@section5.ch>
URL: http://section5.ch/netpp/netpp-htmldoc.tgz

[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/