A *CPSW* Tutorial

October 14, 2016

1 Introduction

The "Common Platform Software" is a package which facilitates communication with common platform firmware. It is intended to provide basic access to firmware features while hiding details of the communication protocols involved. *CPSW* implements abstractions for entities like registers and devices and hides low-level details such as addressing or routing information, byte-order etc.

CPSW uses a description of a device hierarchy – which often is generated by the hardware engineer – in *YAML* format. This description encodes all the low-level information and is interpreted by *CPSW*. *CPSW* generates an internal representation of the hierarchy and offers the user access via the CPSW *User API*.

In this API the hierarchy is represented as a hierarchy of nodes ("*Entries*") and the hierarchy is navigated by means of "Path" objects. All Entries have *names* and can be located by name lookup. Hierarchies are built out of container- elementsi – so called Hubs – and leaf nodes.

Some of the Entries implement *interfaces*. These interfaces offer additional functionality such as reading or writing values, executing commands or reading bulk data.

Because of the limited support for reflection in C++ the user normally must either know what kind of interface an Entry supports (hardware documentation) or inquire by trying to open the desired interface on an Entry.

It is important to note that *CPSW* is essentially *state-less*. This is intentional because complex systems are usually built on top of *CPSW* and software like *EPICS* already offer powerful tools for building and managing stateful systems.

In this tutorial we shall explore the major aspects of *CPSW*:

- Navigating the hierarchy; Paths and interfaces.
- Defining a hierarchy in YAML.
- Using *CPSW* to interact with a remote system.

- Creating a new C++ class which extends *CPSW*'s core functionality.

For the tutorial we will use the *python* binding of *CPSW* but the reader is encouraged to try to reproduce some of the steps in C++ as well.

2 Prerequisites

In order to work through this tutorial the reader is expected to have a working knowledge of

- C++
- python
- YAML
- linux (as an operating-system user; launch commands, execute compilers etc.)

The user should have a fresh copy of this tutorial checked out from `git` and built the version of *CPSW* which comes with it (in the `framework` subdirectory).

You will need a recent toolchain and python3.

2.1 Building *CPSW*

The tutorial comes with its own check-out of *CPSW* – this allows you to browse around and make changes etc. without interfering with other people. Building *CPSW* yourself also familiarizes you with the make process which shall be used again when we create our own C++ class.

First, you need to point the makefile to the locations where the libraries used by *CPSW* are installed

- `yaml-cpp`
- `boost`
- `python3`

For the installation at SLAC, the relevant locations of `yaml-cpp` and `boost` are already defined in the file `config.mak` (in the `framework` subdirectory). However, at the time of this writing the location of `python3` has not been defined and we shall do that now. Unfortunately, the predefined version of `boost` is too old to support `python3` so we will need to point the makefiles to a more recent version.

We will create a new file, `config.local.mak`, (located also in the `framework` subdirectory) from where we can override settings defined in `config.mak`. `config.local.mak` is intended to be used for local modifications which will never be recorded in `git`.

We also define a `INSTALL_DIR` in this file. These are the contents of your new `config.local.mak`

Now you can change back to the tutorial's top directory and type `make install`. The file `INSTALL` provides more information about *CPSW*'s makefile system ([1, 6]).

2.2 Starting `python`

We will now discuss how *CPSW*'s python interface can be loaded. In order for python to be able to find this interface you need to point the environment variable `PYTHONPATH` to the location where `pycpsw.so` is installed (`<top>/bin/linux-x86_64`).

Because *CPSW* requires other libraries (most notably `yaml-cpp` and `boost`) the run-time linker must also be able to find these libraries which is normally accomplished by the linux system administrator configuring the linker's database accordingly ("`ldconfig`").

However, in our case `yaml-cpp` and `boost` are custom built and the linker does not know about them. Therefore you must explicitly "tell" the linker by setting the `LD_LIBRARY_PATH` environment variable.

The file `env.slac` contains suitable definitions for the `bash` shell which you can simply "source" into the shell. You'll need to do this for every session:

```
. env.slac
python3
```

Once the python interpreter is up enter:

```
import pycpsw
```

After this command completes successfully you are able to use the *CPSW* interface. Otherwise you need to make sure

- There were no build errors.
- The `pycpsw.so` file got installed (`make install`).
- The variable definitions in '`env.slac`' are up to date.
- `env.slac` was in fact "sourced", i.e., the variables defined in the file are present in your shell (use `printenv` to check).

2.3 Starting udpsrv

CPSW comes with the `udpsrv` server program which emulates the communication protocols used by *CPSW* and implements emulations of some simple devices. For this tutorial we shall employ udpsrv which means that we do not need to set up special hardware and are independent of network configurations etc.

You need, however, to start udpsrv in order to execute the examples discussed in the tutorial. udpsrv requires the same `LD_LIBRARY_PATH` settings which have been mentioned already. In order to start the server, type

```
bin/linux-x86_64/udpsrv -T 8400
```

The server will open UDP port 8400 where it is listening for requests. Because only a single program may listen on a given port this means that only a single instance of udpsrv may be started using the same port number on any machine. If 8400 is already in use then you have to try a different number. *Note that in this case you will have to modify the YAML definition file (see next section) accordingly.*

3 First Steps with *CPSW*

In this section we load a *YAML*-file which defines a device hierarchy. The *YAML* file `tutorial.yaml` in the `yaml` subdirectory contains definitions which describe the protocol stack implemented by `udpsrv`. If you started the server on a different port then you need to edit the file; look for a `UDP` section and modify the port number to match the `-T` argument.

In python, type

```
>>> import pycpsw
>>> root = pycpsw.Path.loadYamlFile("yaml/tutorial.yaml")
```

`root` now refers to a *CPSW* Path object which represents the root of the device hierarchy.

3.1 Help

The python bindings support basic documentation. For any class or member you can invoke `help(<item>)`:

```
>>> help(pycpsw.Path.loadYamlFile)
Help on built-in function loadYamlFile:

loadYamlFile(...)
<FURTHER OUTPUT NOT QUOTED HERE>
```

or

```
>>> help(root)

Help on class Path in module pycpsw:

class Path(Boost.Python.instance)
<FURTHER OUTPUT NOT QUOTED HERE>
```

3.2 Paths

An important functionality of Paths is the possibility to lookup descendents. Type

```
>>> apath = root.findByName("mmio/folder/hello")
```

You can convert a `Path` to a string representation:

```
>>> print( apath.toString() )
/mmio/folder[0-1]/hello[0-9]
```

The numbers “[0-1]” and “[0-9]” represent *array bounds* which are zero-based and include the last element. This means that there are actually two instances of ‘`folder`’ present under ‘`mmio`’ and and array of 10 ‘`hello`’s in each `folder` which amounts to 20 `hello`s total:

```
>>> apath.getNelms()
20
```

When performing a path lookup without explicitly specifying bounds then the operation will always include *all* instances found.

It is possible to select only a subset of array elements:

```
>>> spath = root.findByName("mmio/folder[1]/hello[3-8]")
>>> spath.getNelms()
6
>>> spath.toString()
'mmio/folder[1]/hello[3-8]'
```

Thus, a Path identifies not only a hierarchical “address” of an entity but also defines multiple levels of array bounds.

A Path can be dupliceted with ‘`clone()`’ and its last element can be stripped with ‘`up()`’:

```
>>> tpath = spath.clone()
>>> tpath.up()
<pycpsw.Child object at 0x7f13d8b977c0>
>>> tpath.toString()
'/mmio/folder[1]'
```

The default lookup operation is not very smart. It does not support features like wildcards or regular expressions. Using the ‘`explore()`’ method it is possible to implement such functionality. `explore()` expects an `IPathVisitor` (see `cpsw_api_user.h`, [2]) callback argument to implement

```
virtual bool visitPre (ConstPath here);
virtual void visitPost(ConstPath here);
```

`explore` recurses through a hierarchy and executes `visitPre()` *prior* to recursing into children and `visitPost()` *after returning* from recursion into children. If `visitPre()` returns `false` then recursion into children is skipped (but `visitPost()` is still called).

`pathGrep.py` is a simple example demonstrating how searching for Paths which match a regular-expression pattern can be implemented. It is left as an exercise to the reader to improve the implementation by avoiding unnecessary recursion (hint: use the `regex` packages’ `partial` feature to check whether the ‘here’ Path could potentially match and let `visitPre` return `False` if this is not the case).

```
>>> import pathGrep as pg
>>> pg.pgrep(root,None)
```

will print the entire hierarchy.

3.3 Entries and Hubs

Every object in the *CPSW* hierarchy is represented by an *Entry*. Entry presents a rather high level of abstraction and provides just enough information for navigating the hierarchy, i.e., `getName()`. For convenience, there is also `getDescription()` which returns a description string – if the designer of the YAML hierarchy has supplied one. Hubs are Entries which can contain a collection of children.

You can obtain a handle to the Entry reached by a by using `tail()` (and the `up()` method which we saw before actually returns the Entry that it strips off):

```
>>> apath.tail().getName()
'hello'
>>> apath.tail().getDescription()
'This is the description of hello'
```

Note that Entries by themselves are not very useful unless they are contained in a Path which uniquely identifies the Entry and also encodes array index ranges (or “bounds”). Very much like a *YAML* alias node the same Entry may be present at multiple places in the hierarchy and only the Path which describes how to reach an Entry contains all the necessary information to perform advanced operations.

3.4 Interfaces of Entries

The really meaningful operations carried out on Entries use *Interfaces*. An interface is an abstract class with a set of defined methods. A given Entry may implement multiple interfaces and conversely: any interface may be implemented by different Entries.

CPSW defines a few commonly used interfaces for accessing integers (*"ScalVal"*), floating-point numbers (*"DoubleVal"*), executing commands or user-defined code (*"Command"*) and accessing bulk data (*"Stream"*). However, additional interfaces may be defined by the user and added to *CPSW*.

An interface is created or "opened" as follows:

```
>>> intrf = pycpsw.<interface>.create( path )
```

where `<interface>` is to be substituted by the name of the desired interface. If the Entry at the tail of the Path actually implements the desired interface then `intrf` will now be a handle to that interface. If the interface is not implemented then a `InterfaceNotImplementedError` is thrown.

Due to the limited support in C++ for reflection it is currently not possible to inquire the number and type of interfaces supported by an Entry. However, in order to do useful work the user must have an idea of the underlying functionality anyways.

We are now ready to try this

```
>>> hello = pycpsw.ScalVal.create( apath )
```

The "hello" array is – on the server – memory backed and probably initialized to random bit patterns.

We can read these back first and then set to the range 1..20 using *ScalVal*'s `getVal()` and `setVal()` methods:

```
>>> hello.getVal()
[4289956834,  548862927, 1795124846,  171206389,
 1363892794,  899570001, 3744639575, 1205068551,
 2941190523, 4152934318, 4241579523, 1726526404,
 2081186026, 3042882507,  316511701,  106114807,
 2428654618, 153782231, 1181565810, 1022839516 ]
>>> hello.setVal( list(i for i in range(1,21)) )
20
>>> hello.getVal()
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

Let us now play with the restricted Path '`spath`' (see above). It only selects elements 3-8 of the second '`folder`':

```
>>> spath.toString()
'/mmio/folder[1]/hello[3-8]'
>>> subHello = pycpsw.ScalVal.create( subHello )
>>> subHello.getVal()
[14, 15, 16, 17, 18, 19]
```

If `setVal()` is given a scalar then all array elements are written with the same value:

```
>>> subHello.setVal(0)
6
>>> hello.getVal()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 0, 0, 0, 0, 0, 0, 20]
```

As the second result shows, indeed only elements addressed by '`spath`' are affected. The return value of `setVal()` also indicates that 6 elements were written.

Array bounds can be further reduced (i.e., without constructing a new Path) by passing '`fromIdx`' and '`toIdx`' arguments to `getVal()` and `setVal()`, respectively. However, only the rightmost index can be reduced with this method.

Note that '`fromIdx`' and '`toIdx`' operate *within* or *based-off* the index range already selected by the Path. E.g., when we say

```
>>> subHello.setVal(88, fromIdx=2, toIdx=4)
3
>>> hello.getVal()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 0, 0, 88, 88, 88, 0, 20]
```

then it can be seen that '`fromIdx`' starts at offset 2 from the (rightmost) index in the path which is 3. Since this path only selects the second '`folder`', there are only 3 elements to be written starting at offset $10 + 3 + 2 = 15$. A single element can be addressed by omitting `fromIdx` (or setting it equal to `toIdx`).

3.5 A look at *YAML*

We encourage you now to open the file `tutorial.yaml` and take a look. Locate the '`hello`' Entry. You will find that it is of class 'IntField'. This is a class which implements the 'ScalVal' interface we had used in the last section. You see that 10 elements ("nelms: 10") of 'hello' are present. The containing '`folder`' (of which 2 elements are present) is of class 'MMIODev'. The latter is a container class which implements memory-mapped addressing. Each of its children is attached at a memory-mapped 'offset'. MMIODev containers may be nested, i.e., a large memory-mapped area may contain sub-areas and the '`folder`' is indeed such a sub-area. An array of two '`folder`'s is present in the top-level '`mmio`' area starting at offset 0. By default, children of MMIODev are 'spaced' by their size, i.e., folder[1] is at offset 0x400.

Now look at the second child of '`folder`'. It is again an 'IntField' – named '`menu`'. It defines the following properties:

```
at: { offset: 0x000001 }
sizeBits: 2
lsBit:    4
enums:
  - { name:  zero, value: 0 }
  - { name:   one, value: 1 }
  - { name:   two, value: 2 }
  - { name: three, value: 3 }
```

This illustrates the following points:

- An IntField may have any bit size (up to currently 64).
- Its 'bits' need not be aligned with a byte boundary (lsBit: 4)
- An optional 'enum' menu can be associated with an IntField. This menu associates integer values with strings.
- Note that '`menu`' starts at byte offset 1 in the containing MMIODev. Legal bit offsets of the least-significant bit are 0..7, i.e., you must anchor the IntField at the correct byte-address and use 'lsBit' only for "sub-byte" shifting.

  Be careful when dealing with IntFields in big-endian layout. The byte-address of the IntField must still be the first byte which overlaps the IntField. E.g., if you have a 12-bit field in big-endian layout with the most-significant bit lined up at a byte boundary e.g., at offset 0 in a MMIODev then the following setup would describe this configuration:

  ```
  at: { offset: 0 }
  byteOrder: BE
  bitSize: 12
  lsBit: 4
  ```

3.5.1 Trying an Enum Menu

After this glimpse we go back to python and manipulate the enum:

```
>>> menu = pycpsw.ScalVal.create(
...               apath.findByName("../menu") )
>>> menu.getVal()
```

```
['zero', 'zero']
>>> menu.setVal( ['one', 'two'] )
2
```

Now try

```
>>> hello.setVal(-1,fromIdx=0,toIdx=0)
>>> menu.getVal()
['three', 'three']
```

Apparently, the memory regions covered by '`hello`' and '`menu`' overlap (this is obvious since `hello` starts at offset 0 with a size of 32-bits (default) and `menu` starts at offset 1).

```
>>> hello.getSizeBits()
32
>>> menu.setVal('zero')
>>> for i in hello.getVal(toIdx=0):
...   print('{:x}'.format(i))
...
ffffcfff ffffcfff
```

The two `menu` bits have now been merged into the `hello` values. Since the byte-offset of `menu` was 1 and the bit shift amounted to 4 bits we find the 2 bits shifted by 12 bits in the 32-bit, little-endian `hello` value.

Note: CPSW does *not* provide any guards against multiple users accessing such overlapping definitions (or accessing multiple registers in the same device etc.). While the underlying SRP protocol executes each transaction atomically, no other protections are implemented and it is the user's responsibility to create and observe mutual exclusion protocols where necessary.

Remember that it is not possible to access only a single `menu` array item with fromIdx/toIdx because these allow only to restrict the right-most index. However, the multiplicity of `menu` has origin in the two `folder` instances – which are at a higher level in the hierarchy. Therefore, you would have to create a new ScalVal interface from a suitable Path object in order to manipulate a single `menu` instance only.

`fromIdx/toIdx` are mainly intended to support indexing large arrays of data residing at the leaves of the hierarchy.

4 Interacting with a Device

It is now time to look at a more realistic example. For that purpose the udpsrv provides a simulation of a "pendulum on a cart" system as depicted in Fig. 1.

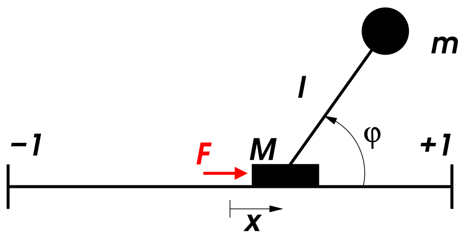

Figure 1: A pendulum (mass m at the end of a mass-less arm of length l) is connected to a cart (mass M) which can move horizontally. An external force F can be applied to the cart. The position of the cart, x, and the angle sustained by the pendulum, φ, can be read from “sensors”.

4.1 Description of the Simulated Target

The simulator provides a few memory-mapped registers which give access to the position readback, the setpoint for the external force as well as some parameters for the simulation. These registers are summarized in table 1. Since 32-bit integer values are used by the simulator to represent numerical values these have been scaled to fit the 32-bit range. The scale factor that must be applied is listed for each register. On readback, a register value has to be multiplied by the scale factor to yield a scaled number and before writing a register the value must be converted by dividing by the scale factor.

The simulation can be re-initialized by writing a position/angle to the respective registers. Note that because multiple 32-bit operations are not carried out atomically the simulator merely *latches* new values for ‘pos’ and ‘iniVelo’ when these are written. Only when ‘phi’ is written then the simulation is re-initialized (using the last latched values of ‘pos’ and ‘iniVelo’).

Likewise, when reading ‘phiRB’ then the simulator simultaneously latches the values for position and time and subsequent reads of these parameters yield the latched values. Thus, in order to obtain a consistent triple read ‘phiRB’ *first*.

All lengths are normalized to half of the horizontal track length which therefore extends from $pos = -1.. + 1$.

4.2 The Beauty of YAML

Compiling the information given in table 1 and writing a driver based on it can be tedious and error-prone. Also, there is the danger that when some of the details change it might not be obvious how to propagate such changes into the driver code.

Last, but not least: once we have a prototype implemented in python and we wanted to proceed with a C++ version then we’d have to port all

Name	Offset	Scale	Signed	Description
vFriction	`0x00`	2^{-32}	N	Viscous damping
gOverL	`0x08`	2^{-16}	N	g/l
mOverM	`0x0c`	2^{-16}	N	m/M
length	`0x10`	2^{-16}	N	Length of pendulum
pos	`0x20`	2^{-31}	Y	Cart position setpoint
phi	`0x24`	$\pi\,2^{-30}$	Y	Pendulum angle setpoint
iniVelo	`0x28`	2^{-15}	Y	Cart+pendulum initial velocity
force	`0x2c`	2^{-15}	Y	External force
posRB	`0x30`	2^{-31}	Y	Cart position readback
phiRB	`0x34`	$\pi\,2^{-30}$	Y	Pendulum angle readback
timeRB	`0x38`	2^{-8}	N	Simulation time readback

Table 1: Simulator register summary. All registers are 32-bit; the scale factor has to be applied when reading (multiply by scale) or writing (divide by scale) register values.

these details.

Luckily, *CPSW* encapsulates most of these steps with the help of *YAML*. In most cases, the hardware-engineer already provides all the detailed information which traditionally would have to be compiled from manuals or header files in standardized *YAML* format.

Since we want to adhere to this workflow the author of the simulator also provided a *YAML* register description of the simulated device in `yaml/pendsimRegs.yaml`. This file contains all the details about the device. The only thing that the system integrator needs to know is where and how the device attaches to the rest of the system.

We have already used the *YAML* description of the udpsrv server, `tutorial.yaml`. From table 2 we extract the base address `0x100400`

Simulated Device	Base Address	Size	TDEST (if streaming)
Bare Memory	0x000000	0x1000	n/a
CPSW Testing	0x001000	0xffe000	1
SPI EEPROM	0x100000	0x0400	n/a
Pendulum Sim.	0x100400	0x0400	44

Table 2: Memory map of the udpsrv server.

of the pendulum simulation. In order to incorporate the register description we need to:

- "#include" the register description *YAML* from the main *YAML* file
- create a new child of the main memory map (`mmio`)
- "merge" the register definition alias

- define the base-address/offset of the simulator

Here are these steps explained in more detail

1. Only the header section up to the first line which does not start with a '#' are scanned for "#include" directives. Make sure there are no empty lines present (but lines containing just a '#' in the first column are fine) ahead of what you are about to add to the header section of `tutorial.yaml`:

   ```
   #include pendsimRegs.yaml
   ```

   Make sure there is no blank space neither before nor after the '#'.

2. Add the following lines to the 'children' of 'mmio':

   ```
           pendsim:
             <<: *pendsimRegisters
             at: { offset: 0x100400 }
   ```

   Make sure the line 'pendsim' has the *same indentation level* as the line containing 'folder'. Also, be careful not to paste the new lines "into the middle" of the lines which make up the definition of 'folder' and its children. The 'pendsim' entry has to be entered either before or after 'folder' and at the same indentation level. Avoid tabs, use only blanks and make sure a blank is present after any ':' in order to separate map keys from values.

More information about the *YAML* syntax and the schema used by *CPSW* as well as the *CPSW*-specific preprocessor syntax can be found in [3, 4].

4.3 Trying it Out

Start a new python session and load the `tutorial.yaml` file as described above. You should now be able to communicate with the simulator. Proceed and create two interfaces '`phiSP`' (setpoint) and '`phiRB`' (readback).

You will see a message about a missing shared object which you can ignore – more about that later.

```
>>> root=pycpsw.Path.loadYamlFile("yaml/tutorial.yaml")
>>> phiSP=pycpsw.DoubleVal.create(
...               root.findByName("mmio/pendsim/phi")
... )
>>> phiRB=pycpsw.DoubleVal_RO.create(
...               root.findByName("mmio/pendsim/phiRB")
... )
```

Since we will be operating with floating-point numbers we instantiated a 'DoubleVal' interface. The readback is read-only and therefore we have to open the read-only interface 'DoubleVal_RO'.

Check out the readback value:

```
>>>phiRB.getVal()
-534217089.0
```

Obviously, *CPSW* has taken notice that this is a signed number (check the *YAML* description which marks this register as 'signed'). All we need to do is apply the proper scale from table 1:

```
>>> phiRB.getVal()/2**30
-0.4992395518347621
```

This result is now in units of π; a look at Fig. 1 confirms that this makes sense. The pendulum has come to rest and hangs down ($\varphi = -\pi/2$).

We can initialize the pendulum to a new position ($\varphi = 0.8\pi$) and verify that it is moving (remember that reading/writing `phiSP`/`phiRB` is what triggers the simulator).

```
>>> phiSP.setVal(.8*2**30)
1
>>> phiRB.getVal()/2**30
0.9794269455596805
>>> phiRB.getVal()/2**30
1.163966953754425e-05
>>> phiRB.getVal()/2**30
-0.9597937744110823
>>> phiRB.getVal()/2**30
-0.13837423734366894
```

4.4 Exercise: Writing the CPSW Interface in Python

The python script '`tutorialGui.py`' uses PyQt4 to display the simulated system in motion. It also provides Gui elements to control the model parameters.

The Gui relies on another script, '`UdpsrvInterface.py`', which implements a thread for periodically reading simulation time, angle and position from the simulated device in order to update the display. It also keeps a dictionary of the parameters which can be accessed via *CPSW*.

`UdpsrvInterface.py` in turn uses a class '`ModelParm`' which does the actual interaction with *CPSW*. The scripts are structured such that this class – which the reader is expected to implement as an exercise – is kept as simple as possible. Each instance of a `ModelParm` object shall represent a single parameter which interfaces to an underlying register from table 1.

1. Create a file '`ModelParm.py`' where you implement the `ModelParm` class.
2. The constructor method (`__init__()`) takes two arguments (in addition to the usual `self`):
   - '`prefix`' which is a reference to a *CPSW* Path object leading to the `pendsim` device which was defined in *YAML*.
   - '`name`', the name of the register (see table 1).

   The constructor should create a 'DoubleVal' (setpoints) or 'DoubleVal_RO' for readbacks, respectively, and store the reference in the `ModelParm`'s `self` handle.
3. Define a '`getDescription()`' method which calls the associated DoubleVal's `getDescription()` method and propagates the returned description string.
4. Define a '`getVal()`' method which calls the associated DoubleVal's `getVal` method, applies the appropriate scale factor from table 1 and returns the scaled value.
5. Define a '`setVal()`' method which takes (in addition to the usual `self`) a single (float) argument which needs to be scaled by the factor from table 1 and passed down to the associated DoubleVal's `setVal()`.

It is worthwhile studying the section of the constructor of the `Model` class (in `UdpsrvInterface.py`) where the `ModelParm` objects are instantiated:

```
for child in self.modl.tail().isHub().getChildren():
  nam             = child.getName()
  self.parms[nam] = ModelParm.ModelParm( self.modl, nam )
```

As you can see, the code iterates over the children of `modl` (which is a reference to the `pendsim` device), retrieves their names and creates a `ModelParm` for each of them. This is an example for how GUI elements can be created in an automatic fashion from *CPSW*.

4.5 Launching the GUI

You can now launch the Gui and test your `ModelParm` class. Remember that the environment variables discussed earlier need to be set. The scripts require python3:

```
python3 tutorialGui.py
```

The Gui should be started from the tutorial top directory in order to locate dependent *YAML* and python files (including your `ModelParm.py`).

5 Working with Streams

A stream provides direct access to bulk data on a remote device. Data are presented to the user after passing through the (optional) RSSI, packetizer and TDEST demultiplexing layers as a sequence of raw datagrams (i.e., *CPSW* does not further interpret or manipulate the data and passes them "as-is" on to the user).

The pendulum simulator device does implement a stream where it feeds position and angle readings. I.e., instead of periodically polling the current position readback our application may '*listen*' to a stream.

The simulator provides (unsolicited) readings of the simulation time, the pendulum angle and the horizontal position over a streaming interface. The data format of the streamed message is shown in table 3. The time, angle and position are stored in *little-endian* byte-order as 64-bit, fixed-point numbers which can be converted into floating-point representation by division by 2^{32}. Note that this scale factor is common to all elements and *different* from the scales in table 1.

Address	LSB, Byte 1, Byte 2 ... Byte 7		Byte 8 ...	MSB
0x00	Time	(fractional part)	Time	(integer part)
0x08	φ	(fractional part)	φ	(integer part)
0x10	x	(fractional part)	x	(integer part)

Table 3: Layout of the stream datagram containing simulator state information. The 64-bit numbers are a fixed-point representation with the radix point located between bits 31 and 32.

5.1 Defining a Stream in *YAML*

The first thing we need to do is to add a definition of the streaming Interface to our *YAML* file.

The first questions are "where is this definition to be added" and "what information is required"? We might be tempted to answer the first of these questions by proposing to add the stream definition to the "pendsim" device definition.

However, this device describes a memory-mapped entity and its register details etc. – but the streaming channel is actually rather independent from the device registers (similar to an interrupt line of a PCI device which can be routed independently from PCI on a motherboard).

The streaming channel is actually originating directly at the *communication device*. (Again the analogy of the interrupt line which may be connected directly to a central interrupt controller.)

In fact – the details of the stream communication are dominated by choices made by the system integrator rather than the device designer. A stream can use a separate communication channel (e.g., UDP port) altogether or share parts of the communication stack with the memory-mapped devices. For this reason, a streaming interface can often not entirely be defined by the device-specific *YAML* file.

In the case of our example the stream actually shares many protocol layers but is eventually demultiplexed based on a “TDEST” identifier: `44`. The main `tutorial.yaml` file defines a ‘StreamProtoConfig‘ alias node with all the common definitions applicable to streams (as implemented by udpsrv) except for TDEST.

Now open the `tutorial.yaml` file and add the following definition *as a sibling to ‘mmio’* (again: pay careful attention to indentation levels and avoid TABs):

```
pendsimStream:
  class: Field
  at: {       <<: *StreamProtoConfig,
        TDESTMux: { TDEST : 44 }     }
```

5.2 Reading from a Stream

With the streaming interface now defined in *YAML* we can proceed to exercise the stream. Begin with loading the `tutorial.yaml` file into python (see section 3). A streaming interface is – like any other interface in *CPSW* – opened from a Path:

```
>>> strm=pycpsw.Stream.create(
...             root.findByName("pendsimStream")
... )
```

The python bindings require an object which supports the (so called “new”) buffer interface (consult the python documentation for more information) to be passed to the stream’s ‘`read()`‘ (or ‘`write()`‘) method. In python3 we may use an `array` object (`numpy.ndarray` is another option). Since we are dealing with three 64-bit numbers we create an array of quads:

```
>>> import array
>>> data=array.array('q', (0 for i in range(0,3)))
```

and read from the stream (the return value is the number of bytes read):

```
>>> strm.read(data)
24
```

Note that the `data` array now contains the raw data. If the host computer does not use 'little-endian' representation then the raw data need to be byte-swapped.

We extract the angle and convert to a floating point number (see table 3):

```
>>> phi = float(data[1])/2.**32
-1.5706
```

The pendulum is apparently hanging in its equilibrium position.

5.2.1 A "Strange Phenomenon"

From the same session that uses the streaming interface you could now try to read the angle from the corresponding readback register:

```
>>> phiRB = pycpsw.ScalVal_RO.create(
...                root.findByName("mmio/pendsim/phiRB")
... )
>>> phiRB.getVal()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
pycpsw.IOError: No response -- timeout
```

After issuing the `getVal()` command nothing happens for a while and eventually a timeout error occurs. What is going wrong? The cause for this behavior is the configuration of the communication stack which *shares* the *RSSI* (reliability) layer between the stream and register access. RSSI guarantees reliable in-order delivery of all traffic. Because we are not consuming all the streaming messages as they arrive the output queue of `pendsimStream` fills up and prevents RSSI from delivering more messages. Since RSSI must deliver *all* messages in order (the demultiplexing of streaming messages and register transactions is done at a higher protocol level) it eventually stalls (and puts back-pressure on the streaming data producer). However, as a side-effect, register transactions are also stalled.

To remedy the situation we can *close* the streaming interface. In the C++ API this is achieved by releasing all shared pointers to the stream in question. In python this is not necessarily enough since garbage collection may happen at a later time but the Cpython implementation apparently does release the underlying shared pointer immediately:

```
>>>del(strm)
>>>float(phiRB.getVal())/2.**30*3.141592659
-1.5707972482125554
```

The lesson to be learned here:

- Only instantiate a streaming interface if you are consuming data.

- Consume all streamed data.
- Release all shared pointers to a streaming interface when you no longer need it.
- In python you should avoid creating unused streams; closing may depend on unspecified/implementation-defined semantics.

5.3 Exercise

As an exercise you should write a small python class which "glues" the stream fed by udpsrv into `UdpsrvInterface.py`.

1. In the tutorial's top directory create a file '`StreamHandler.py`' where you define a '`StreamHandler`' class.
2. The constructor (`__init__()`) takes as a single argument (in addition to `self`) a Path representing the root of the *CPSW* hierarchy.

   The constructor shall open/create the streaming interface and store its reference in `self`. Also, an array object with three 64-bit elements shall be pre-created and attached to `self`.

   Optionally, a boolean flag shall be set (in `self`) indicating whether byte-swapping is required.
3. The class shall define a `read()` method which takes a single (in addition to `self`, that is) numerical parameter which communicates a timeout (in micro-seconds) for the read operation.

   The `read()` method shall attempt to read from the stream into the buffer-array for up to the specified timeout.

   It the read-operation on the stream times out then the method shall return `None`.

   If the read-operation is successful then the method shall return a tuple with three floating-point values: $time$, $angle$ and $position$ in that order. These values shall be properly scaled (see table 3).

   Optionally, byte-swapping shall be performed – if required as determined by `__init__()` – on the array elements (before converting them into floats).

Use [2] and/or python `help()` for more information about the Stream interface's `read()` method as well as other details you might need to know.

Test this class. Below the graphical window in the Gui the status line should indicate that streaming is used.

6 Writing a *CPSW* C++ Extension

In this section we shall get an idea of how *CPSW* may be extended at the C++ level. In most cases this means extending a subclass of EntryImpl (the implementation of Entry) in order to implement one of the base interfaces for a different kind of underlying hardware, e.g., a custom implementation of a 'Command' [5]. Such a class may also introduce new interfaces (consult the 'FirmwareLoader' application – which is *not* part of this tutorial, however – for an example [7]).

CPSW provides a number of hooks for such an extension:

- The extension class should be compiled into a *shared object*.
- The extension, when run-time linked into a *CPSW* application *registers* itself with *CPSW*.
- A *YAML* definition may list an extension among the elements of the 'class' property of a node in the hierarchy.
- When building the hierarchy, *CPSW* examines the 'class' propery of each node it has to create. It performs a lookup of the names listed in the 'class' property in its registry of built-ins and extensions. If the desired 'class' is found then creation is delegated to a factory method of the class.
- If a desired 'class' is *not* found then *CPSW* tries to dynamically link a shared object with the same name. If this operation is successful then it triggers registration of the new extension (see above). *CPSW* will then – on a second attempt – find the desired 'class' and subsequently execute its factory method to create the new node.

6.1 Extension Design Goal

With vanilla *CPSW* we have not been able to transparently use the scale factors. Although they are present in the *YAML* register definition the built-in class (`IntField`) does not make use of them. Instead, we had to *manually* copy them into our python class (`ModelParm`) and apply them when interacting with *CPSW*. It would obviously be preferable if *CPSW* could "do the right thing" under the hood.

Let's therefore create a *CPSW* extension class which applies a linear transformation when converting an integer to a floating-point value. The extension should be able to extract the conversion parameters from *YAML*.

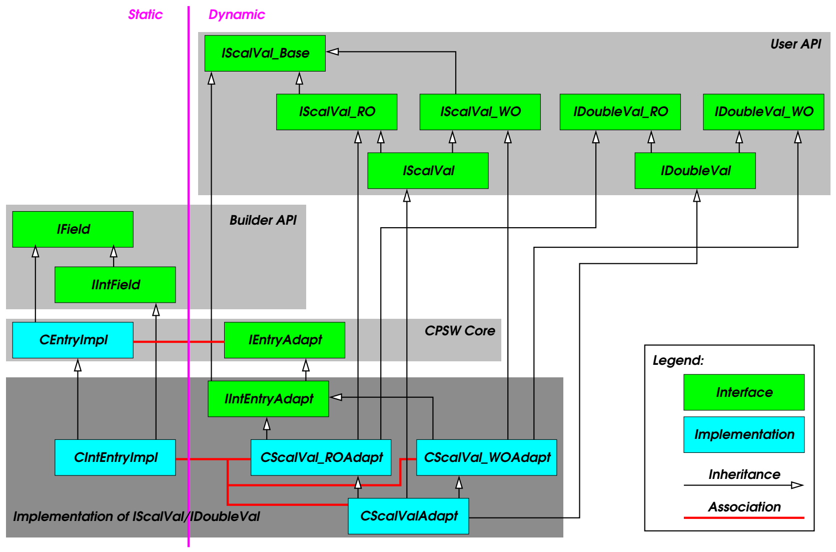

Figure 2: Hierarchy of Classes and Interfaces for the CIntEntry-Impl/CScalValAdapt_XX implementation of ScalVal/DoubleVal and related interfaces. The "static" class CIntEntryImpl is associated with Adapters CScalValXXAdapt.

6.2 Extending an Existing Class

The first decision we have to make is: *what existing class should we extend*? Since `IntField` already provides most of the functionality we need it is natural to use it as a starting point. `IntField`, however, is itself an abstraction. The actual hierarchy of classes is a bit more complex and we shall return to this in a moment. But first we explain the concept of a *CPSW* "*Adapter*" class.

The purpose of an Entry in most cases (including this one) is providing an *interface* for user interaction. As it has been mentioned already, the Entry itself is *not* holding the full information required by an interface; it is a rather "static" entity and like the *YAML*-node from which the Entry was created it may be present at multiple places in the hierarchy. An *Adapter* is a object which contains dynamic information (such as the Path from which an *interface* is created) and it "glues" the implementation of an *interface* to an *entry* ("Association" in Fig. 2). Thus, an implementation of an Entry in *CPSW* usually comes as a *pair* or even *set* of classes: one class which implements the static aspects and an *Adapter* class for each supported *interface* (alternatively, there could also be a single *Adapter* which implements multiple *interfaces* – e.g., `CScalVal_ROAdapt` in Fig. 2 implements IScalVal_RO as well as IDoubleVal_RO).

The hierarchy of implementation- and interface-classes is depicted in Fig. 2.

The class CIntEntryImpl and its associated *Adapter*s (dark grey box at the bottom left of Fig. 2) are defined in the files `cpsw_sval.h` and `cpsw_sval.cc`. CScalVal_ROAdapt has a virtual `int2dbl()` method for conversion of integers to floating-point values and CScalVal_WOAdapt declares a virtual `dbl2int()` method for the inverse operation. The default implementation just performs a one-to-one transformation. These methods are thus obvious candidates to be overridden, extending the functionality from a one-to-one relation to an arbitrary linear transformation with configurable "scale" and "offset" parameters. These parameters are to be extracted from *YAML* and this is the job of an associated class, CInt2Dbl, which derives from CIntEntryImpl. Fig. 3 shows the new classes and how they derive from existing ones.

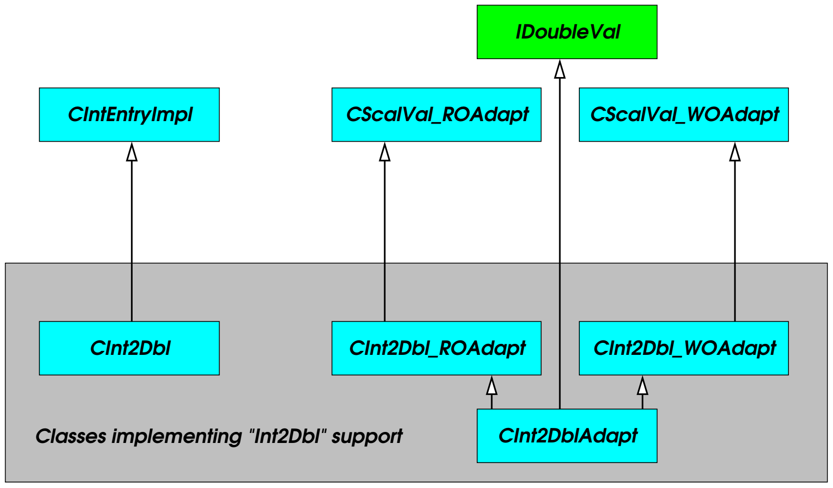

Figure 3: New classes to support integer to floating-point conversion. The adapters CInt2Dbl_ROAdapt and CInt2Dbl_WOAdapt override the actual conversion methods. CInt2Dbl holds the conversion parameters (scale/offset) which are extracted from *YAML*.

6.3 Implementation

Study the implementation in the files `int2dbl.h` and `int2dbl.cc`. The header reflects the inheritance relationship depicted in Fig. 3.

The CInt2Dbl class adds members for scale and offset to what CIntEntryImpl already provides. There is some boilerplate code:

- Every subclass of Entry should implement a copy constructor and a `clone()` method which invokes this copy constructor. `clone()`

receives a special *Key* argument which is a guard against direct execution of `clone()` by the user. `clone()` must only be used by *CPSW* internally since the return value must be properly wrapped into a *shared pointer* before it is handed to the user. *Key* objects cannot be created publicly and since a *Key* is required to call `clone()` the *Key* object helps to assert fine-grained access control. You'll find this idiom at several places in *CPSW*.

- Every subclass of Entry must define a `static _getClassName()` method which returns the (unique!) name of the class as it is to be used by the "class" property in *YAML*. The shared object into which the new class (along with its *Adapters*) is to be built *must bear the same name* so that it can be located by the dynamic linker.

- A virtual method `getClassName()` must be added. It should just call `_getClassName()` so that the class name can be obtained in a polymorphic fashion (the `static` version is required by the class registry – since no object of the class exists *a-priory* the registry must use the `static` method).

- Every subclass of Entry must provide a constructor which takes a *Key* and a YamlState argument. This is what allows construction of objects from *YAML*. The constructor extracts all relevant parameters from YamlState with `readNode()`.

- `dumpYamlPart()` is responsible for storing member values into a *YAML* node permitting a hierarchy which was created with the builder API to be saved in *YAML* format. It does the inverse operation of the constructor.

- `createAdapter()` is the factory method which creates an object of the Entry's associated *Adapter* class. It is being passed a `std::type_info` reference which identifies the *interface* the user wants to create. The method should inspect this argument and decide if any *Adapter* implements the desired interface and create a new *Adapter* object if this is the case. Otherwise, the method should chain to the superclass.

- Every subclass of Entry should add itself to the *CPSW* class registry by expanding the macro `DECLARE_YAML_FIELD_FACTORY()`. Note that this macro must be expanded from a file which is being linked (not an issue if shared libraries or shared objects are used; the note applies to the situation where the extension classes are residing in a static library).

The most interesting pieces are the constructor from *YAML* and the `createAdapter()` method.

The methods `int2dbl()` and `dbl2int()` which compute the actual conversions are rather trivial. Note how they fetch the conversion parameters from the associated CInt2Dbl object.

It is also noteworthy that the conversion routines in this example do *not* make use of any information held by the *Adapter* class. Thus, it would in theory be possible to forego the business of *Adapters* – the converters could just have been members of CIntEntryImpl. However, imagine you would want to implement a similar class but this time the scale factor should be read from another ScalVal somewhere in the hierarchy – the current design would allow you to do that and you would then store the necessary Path and/or ScalVal references in the *Adapter* object.

6.4 The "makefile"

Add the following lines to the `makefile` in the tutorial top directory. This will result in a shared object `Int2Dbl.so` being built (after typing `make`).

```
PROGRAMS         += Int2Dbl.so
Int2Dbl_so_SRCS += int2dbl.cc
Int2Dbl_so_LIBS  = $(CPSW_LIBS)
```

For more information about the *CPSW* makefile system consult [6, 1].

6.5 Cleaning Up "ModelParm.py"

If you look at `pendsimRegs.yaml` again then you will see that the 'class' property of the individual registers is actually a *sequence* which lists multiple classes which could possibly support the registers. Since we hadn't actually built 'Int2Dbl' *CPSW* was unable to locate this class and fell back on a regular 'IntField'. Now, however – assuming that the LD_LIBRARY_PATH is set up correctly – the dynamic linker should be able to locate our new Int2Dbl.so and use it according to the algorithm explained above.

Since all of the scaling is now taken care of transparently you can copy the ModelParm.py file to a new version: ModelParmScaled.py. Open this new file in your editor and remove all the code which does compute the scaled values. The ModelParm class should be trivially simple now.

Next, edit UdpsrvInterface.py and close to the top modify the `import` command to use ModelParmScaled.py instead of ModelParm.py:

```
import ModelParmScaled.py as ModelParm
```

Launch the TutorialGui.py again and verify that it is working.

7 Further Reading

References

[1] framework/INSTALL[.html],
"HowTo" build *CPSW*.

[2] framework/cpsw_api_user.h,
The *CPSW* "user" API.

[3] http://www.yaml.org/spec/1.2/spec.html,
Specification of the *YAML* syntax.

[4] framework/README.yamlDefinition[.html],
description of *YAML* as used by *CPSW*; definition of preprocessor directives.

[5] framework/test/cpsw_myCommand.cc,
example for implementation of a custom command.

[6] framework/makefile.template,
template for a makefile using *CPSW*'s makefile system.

[7] FirmwareLoader,
A *CPSW* application which adds new classes to *CPSW*. Residing in its own `git` repository
/afs/slac/g/cd/swe/git/repos/package/cpsw/FirmwareLoader.git