TPM Access Layer

Alessio Magro

This document serves as a rough user-guide for the *TPM Access Layer* software tools. These include a C-like API for communicating with TPMs, a Python wrapper with extended functionality for interactive debugging and scripting, as well as a C++ server for communication with higher software layers. The latter program is primarily designed to be used by the Web-based UI application, and is based on a Protobuf over ZeroMQ interface. This document is not meant to provide a detailed technical discussion on the software's implementation. This access layer is still in active development, and many features are not yet implemented (such as loading of firmware onto the FPGAs, access to on-board SPI devices, and so on). This document will be continually updated as new feature are introduced.

At the core of all these features lies an XML schema which describes the memory maps of loaded firmware and the board itself. This provides a map between register names or mnemonics to a memory address on the board's address space. In the UniBoard Control Protocol (UCP), all register and memory accesses are performed on a provided memory address, such that any address conversion has to be performed on the client side. For this reason, all checks (such as memory block length, bitmasks, access permissions ...) have to performed on the client side as well. The XML file must contain all this information. Section 1 describes the XML schema in detail.

The access layer API can be used directly by custom C or C++ code. A C-like interface was chosen so as to make it possible to use this API from pure C-code, although the library itself is written in C++. The library has no external third-party dependencies, and is easily compilable with a modern g++ compiler (currently version 4.8+). Section 2 provides additional details on this layer. However, it is generally tedious to debug and test functionality using compiled C code, and for this reason a Python wrapper is also provided. This loads the compiled C++ library and provides direct access to the underlying API. The wrapper extends the functionality of the underlying library by providing Pythonesque behavior. Section 3 describes this wrapper in further detail.

1 XML Memory Map

The XML file provides a mapping between register and memory block names/mnemonics to memory address on the board. In the TPM's case, the XML file can contain the mapping for up to three 'devices': the CPLD (or board), the firmware loaded on the first FPGA and the firmware loaded on the second FPGA. The same firmware can be loaded on both FPGAs. The register in a firmware map can be logically split into components (such as channeliser, beamformer, calibrator ...), and in turn each register can be composed of multiple bit-fields, where disjoint sets of bits have different interpretation in the firmware. For this reason, an XML mapping file is composed of a hierarchy of four nodes (excluding the root node):

Device node Highest level node, representing the board itself (CPLD) or the firmware

loaded on the FPGAs. The identifiers for nodes in this level are "CPLD", "FPGA1" and "FPGA2".

Component node Registers for each device can be grouped into components, which makes it easier to logically partition the functionality of these registers. For example, registers belonging to different firmware components can be grouped into different nodes at this level. A base address can be specified for each component.

Register node Each component can be composed of multiple registers or memory blocks, each requiring an identifier (the name or mnemonic) and the memory-mapped address. Additional attributes can also be specified, such as the size in words, access permission, the bitmask and a description.

Bit node A register can be composed of a combination of multiple bitfields. These bitfields are specified in the fourth node level, having an entry for each bitfield in the register. A bitmask per bitfield must be specified.

The XML listing below provides a very concise example of an XML mapping file. Each of the node attributes are defined as follows:

id The device, register, memory block or bitfield name/mnemonic. This does not need to be unique within the XML file (such that the same firmware can be loaded on both FPGAs). The IDs of successive levels in the hierarchy are combined to generate the full name in the access layer. For example, the full name of the first bit in `bit_register`, `regfile`, `FPGA1` in the example below would be `regfile.bit_register.bit1` (the registers are partitioned internally by device).

address The memory address to which the register is mapped. A base address can be specified at all levels (except bitfields). In this case, successive base addresses are added together to form the final address.

permission Access permissions for the register or memory block. Allowed entries are `r`, `w` and `rw`.

size The size in words of the register or memory block. If this value is not specified, then the register is assumed to be of size 1 (32-bits). Memory blocks can essentially be defined by setting this value. Note that if a bitmask is specified for a memory block with size greater than 1, then it is applied for all consecutive words.

mask The bitmask to be applied to the values after reading from or before writing to the memory address.

description A textual description for the register.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<node>

  <node id="CPLD">
    <node id="regfile" address="0x00000000">
      <node id="register1" address="0x00000010" permission="rw"
            size="1" mask="0xFFFFFFFF" description="..."/>
```

```
      ...
    </node>
    ...
  </node>

  <node id="FPGA1" address="0x100000000">
    <node id="regfile" address="0x00001000">
      <node id="bit_register" address="0x00000010" />
        <node id="bit1" mask="0x00000001" permission="rw"
              description="..." />
        <node id="bit2" mask="0x00000002" permission="rw"
              description="..." />
        ...
      </node>
      ...
    </node>
    ...
  </node>

  <node id="FPGA2"> ... </node>

</node>
```

Listing 1: XML memory map

2 TPM Access Layer API

The TPM Access Layer provides a C-like API for communicating with TPMs using UCP. Although monitoring and control of TPMs is the primary use-case for this API, its software design allows the same layer to be usable with other FPGA boards. Minimal changes are required to make it compatible with UniBoards (since they use the same protocol), whilst custom Protocol and Board classes are required to make it compatible with other boards. Details on how to change and extend this layer will be provided in a separate technical guide. The listing in Appendix B lists the currently defined API calls in this layer. The access layer is capable of handling connections to multiple boards, which could be of different types.

3 Python wrapper

The Python wrapper can be used for interactive testing and scripting. All the functionality is defined in the package `tpm`. This section will provide a rough tutorial on how to use this package. In the following examples, the wrapper will be connecting to one TPM board and issue requests via the underlying access layer. Before these interactions can occur, two operations need to be performed: connect to the board and load the XML memory map. The listing below shows how to do this.

```
from tpm import *

# Create a new TPM instance
tpm = TPM(ip="127.0.0.1", port=10000)
```

```
# Call load firmware to load XML file
tpm.loadFirmwareBlocking(Device.FPGA_1, "/path/to/xml/file")
```

The first line imports all definitions from the `tpm` package. The next statement then creates a new instance of call `TPM`. When an IP and port are specified, as in this example, the `connectBoard` in the library is called, setting up all required internal data structures. It will also check to see whether the destination IP and port can process UCP requests, and if not, an error is generated. For this reason, either a TPM is required, or a script which acts as a UCP server. You can get this script from the `scripts` directory in the code repository, called `mock_tpm.py`. Just run this on a separate console, and it will bind itself to all local IP address and wait for read and write requests. The `TPM` constructor also accepts an additional argument, the filepath of the compiled access layer, in cases where the library is not directly accessible by the wrapper. This can be passed with `library='path'`.

Once connected, the XML memory file needs to be loaded. The way in which this will be performed on the board is not yet defined, so for the time being the `loadFirmware-Blocking` call must be used. This takes the device on which the "firmware" will be loaded and the path to the XML file. `Device` is a representation of the `DEVICE` enum in the library, and can currently have three values: `Device.FPGA_1`, `Device.FPGA_2` and `Device.Board`. Only the two former cases are allows for this functional call, since you cannot load a firmware on the board itself. This call will also populate all internal data structures containing register and memory block mapping information, such that read and write operations can immediately be issued by the user.

The wrapper provides two ways in which registers can be read from or written to, once based on direct function calls and the other using indexed access. The following listing provide the function prototypes for using the functions directly:

```
# Read register values from specified register
# device   - Device to which this register belongs to
# register - Register name/mnemonic
# n        - Number of words to read
# offset   - Address offset to start reading from
# Returns: Value or list of values
def readRegister(device, register, n = 1, offset = 0)

# Write values to specified register
# device   - Device to which this register belongs to
# register - Register name/mnemonic
# values   - Value or list of values to write
# offset   - Address offset to start reading from
# Return:  Success or Failure
def writeRegister(device, register, values, offset = 0)

# Read values from specified memory address
# address  - Memory address
# n        - Number of words to read
# Returns: Value or list of values
def readAddress(address, n = 1)

# Read register values from specified register
# address  - Memory address
# values   - Value or list of values to write
# Returns: Success or Failure
```

```
def writeAddress(address, values):
```

The two functions defined above require both the `device` and `register` name to be able to compute the memory address. In all cases, checks are performed by the wrapper to make sure that request don't go out of bounds (for example, the length of the list of values to write are larger than the size of the register, the register has write permissions in case of writes, and so on). The list of registers which have been loaded can be displayed either with the `listRegisterNames()` function call, or with the following statement: `print tpm`, where tpm is a `TPM` class instance. This will print out the list of register with additional information, as shown below:

```
 Device   Register                                Address      Bitmask
______   ________                                _______      _______
FPGA 1   regfile.block2048b                      0x1000L      0xFFFFFFFF
FPGA 1   regfile.date_code                       0x0L         0xFFFFFFFF
FPGA 1   regfile.debug                           0x10L        0xFFFFFFFF
FPGA 1   regfile.jesd_channel_disable            0xcL         0x0000FFFF
FPGA 1   regfile.jesd_ctrl.bit_per_sample        0x4L         0x000000F0
FPGA 1   regfile.jesd_ctrl.debug_en              0x4L         0x00000001
FPGA 1   regfile.jesd_ctrl.ext_trig_en           0x4L         0x00000002
FPGA 1   regfile.reset.global_rst                0x8L         0x00000002
FPGA 1   regfile.reset.jesd_master_rstn          0x8L         0x00000001
Board    regfile.ada_ctrl.ada_a_ada4961          0x30000010L  0x00000008
Board    regfile.ada_ctrl.ada_fa_ada4961         0x30000010L  0x00000008
Board    regfile.ada_ctrl.ada_latch_ada4961      0x30000010L  0x00000008
... (redacted)
```

Indexed access provides a more Pythonesque methods of accessing registers. The three statement below will perform the same operation. All of them will read all the values for memory block `regfile.block2048b`, which is defined for `FPGA1`. The first statement specified the full register name as provided by the access layer. In cases where the same full name is specified for both FPGAs (same firmware is loaded) the device name can also be specified as in the second example. In the third statement, the memory address is provided immediately, such that the `readAddress` function is called instead of `readRegister`.

```
1. tpm['regfile.block2048b']

2. tpm['fpga1.regfile.block2048b']

3. tpm[0x1000]
```

Similar statement can also be defined for writing values. In the examples below, the memory region define for regfile.block2048b (512 words) will be initialized with `1` for all values. Note that in order to write to an address offset, the function calls must be used directly.

```
1. tpm['regfile.block2048b'] = [1] * 512

2. tpm['fpga1.regfile.block2048b'] = [1] * 512

3. tpm[0x1000] = [1] * 512
```

The wrapper and underlying library also handle bitmasks automatically, as shown in the example below, where `regfile.bitregister.bit8` at address `0x10001000` has a bitmask of `0x00000008`

```
1. print tpm['regfile.bitregister.bit8']        Output = 0

2. tpm['regfile.bitregister.bit8'] = 1

3. print tpm['regfile.bitregister.bit8']        Output = 1

4. print tpm[0x10001000]                        Output = 8
```

Reads from and writes to a bitfield will automatically be shifted such that the value belonging to the bitfield itself are displayed. Writes to a bitfield are performed as a read-modify-write operation in the access layer library. The Python wrapper also provides some additional features, as show below:

```
            # Use regular experssions to search for a particular register
            # from list of register, returning a dictionary for each
            # match. Can also print out the results
Statement:  tpm.findRegister("*block2048b*", display = True)
Output:     regfile.block2048b:
            ----------------------
            Address:        0x1000L
            Type:           RegisterType.FirmwareRegister
            Device:         Device.FPGA_1
            Permission:     Permission.ReadWrite
            Bitmask:        0xFFFFFFFF
            Bits:           32
            Size:           512
            Description:    For testing

            # Return number of registers
Statement:  len(tpm)
Output:     32
```

Once all operations have been performed on a TPM, the the script should disconnect from the board:

```
tpm.disconnect()
```

Appendix A - Compilation and Installation

The Access Layer source code can be retrieved from `https://github.com/lessju/TPM-Access-Layer.git`. The top directory contains four directories:

- `doc`, which contains some documents, example XML files and this user guide
- `python`, which contains the Python wrapper and setup scrip
- `script`, which contains helper scripts
- `src`, which contains the source code for the library and server

The `src` directory in turn contains three directories, one containing the source code for the C++ library (`library`), one containing the source code of the server (`server`) and one containing source code for exporting the library as a Windows DLL (experimental, `windows`). Each directory contains a Makefile which can be used to build and install each tool.

Compiling the library

Requirements: g++ version 4.0+. Compilation and installation steps:

1. `export ACCESS_LAYER=/path/to/top/level/access/layer/dir`
2. `cd $ACCESS_LAYER/src/library`
3. Edit `INSTALL_DIR`, `LIBRARY_NAME` and `GCC` in the Makefile so that it can be compiled and installed on your system. Note that `GCC` should point to a C++ compiler
4. `make`
5. `sudo make install`

Installing the Python Wrapper

Ideally, the python version should be 2.5 or higher. Older versions can also be used, however the `ctypes` package will need to be installed.

1. `sudo pip install enum34` (Assuming pip is installed)
2. `export ACCESS_LAYER=/path/to/top/level/access/layer/dir`
3. `cd $ACCESS_LAYER/python`
4. `sudo python setup.py install`

Compiling the server

The server uses two third party libraries: ZeroMQ and Protobuf (latest versions of both). You will need to install these in order to use the server.

1. `export ACCESS_LAYER=/path/to/top/level/access/layer/dir`
2. `cd $ACCESS_LAYER/src/server`

3. `protoc --cpp_out=. message.proto`. This only needs to be performed once, or when message.proto changes

4. Edit `LIBRARY_DIR` and `GCC` in the Makefile so that it can be compiled on your system.

5. `make`

If the access layer library was not installed in a system directory, then update `LD_LIBRARY_PATH` to point to the library, otherwise the server won't manage to load the library: `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/library`.

Appendix B - Access Layer API

```
// Connect to a board with specified IP and port. Returns unique
// board identifier
ID connectBoard(const char* IP, unsigned short port);

// Disconnect from board with ID id
RETURN disconnectBoard(ID id);

// Reset board (currently not implemented)
RETURN resetBoard(ID id);

// Get board status (currently not implemented)
STATUS getStatus(ID id);

// Get a list of board and firmware registers. A memory map needs
// to be loaded.
REGISTER_INFO* getRegisterList(ID id, UINT *num_registers);

// Read register value from specified device/register. Number of
// values and address offset can be specified.
VALUES readRegister(ID id, DEVICE device, REGISTER reg, UINT n,
                    UINT offset = 0);

// Write register value to specified device/register. Number of
// values and address offset can be specified.
RETURN writeRegister(ID id, DEVICE device, REGISTER reg, UINT n,
                     UINT *values, UINT offset = 0);

// Read a number of values from address. To be used for debug mode.
VALUES readAddress(ID id, UINT address, UINT n);

// Write a number of values to address. To be used for debug mode.
RETURN writeAddress(ID id, UINT address, UINT n, UINT *values);

// Load (non-blocking) firwmare to device. Behaviour currently unspecified
RETURN loadFirmware(ID id, DEVICE device, const char* bitstream);

// Load (blocking) firmware to device. Behaviour currently unspecified.
RETURN loadFirmwareBlocking(ID id, DEVICE device, const char* bitstream);
```

Listing 2: TPM Access Layer API listing

`ID`, `VALUES`, `UINT` and `DEVICE` are defined in *Definitions.hpp*. This header file also contain a macro to enable or disable INFO output during execution.