PyFabil

Alessio Magro

This document servers as a rough user-guide for PyFabil and related software tools. These include a C-like API for communicating with digital boards, 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 high level client applications 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. The tools are still in active development, and many feature are not implemented yet. This document will be continually updated as new features are introduced.

1 Introduction

The primary aim of pyfabil is to provide a single access points to multiple digital boards. In this way, the same function signatures can be used to access any board type and instance. All communication protocol as well as device and register handling details are hidden to the client. The end user does not need to know whether the underlying communcation is performed using a UDP or TCP-based protocol, or whether the list of registers is loaded through an XML file or from a the programmed firmware itself. Most of these are implemented in a C++, primarlity for speed. The higher level Pyhon interface provides an abstracted view of the board.

Three boards are currently supported:

- UniBoard
- ROACH and ROACH2
- iTPM (Italian Tile Processing Module, once of the candidates for the SKA LFAA digital backend)

Figure 1 depicts the overall design of the Python layer. TThe interface block encapsulates the API provided by the FPGA Board Access Layer shared library, allowing higher level components to call C functions directly. This can be used on its own, allowing software developers to implement their own abstract functionality. The `Defs` block is a place-holder for package-wide declarations, including enumeration, decorators, Exceptions, and so on. The `FPGABoard` is an abstract superclass which implements common functionality, available to all board types. These include :

- Connecting to and disconnecting from a board
- Load and unload plugins

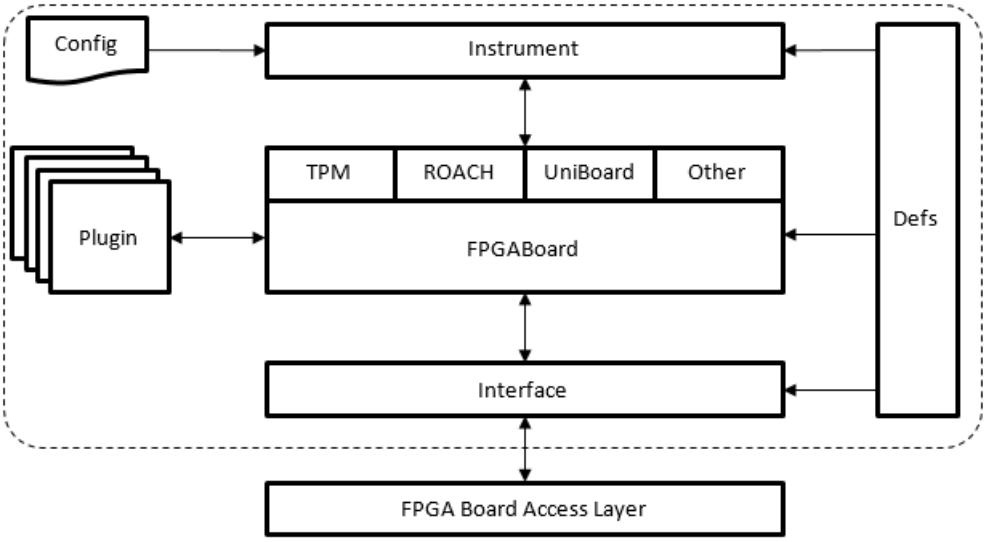

Figure 1: PyFabil overall design

- Calling the wrapped C++ functions with default parameters
- Keeping the internal state of the instance (for example, FPGA(s) programmed, connected, faulty)
- Provide helper functions, such as to search for a particular register or device name
- Automatic handling of register bitfields (all masking and shifting is performed internally if this information is available)
- Significant error checking and logging routines

For each new board type an new class should be implemented which inherits from `FPGABoard`. Any method defined in the superclass can be re-implemented (overridden), however one should ensure that any book-keeping defined in the parent's class are performed here as well. Ideally, the parent's method is called directly after board-specific functionality is implemented. This generally takes the form of: `super(ClassName, self).method_name()`.

When firmware is loaded onto FPGAs the functionality of the board changes. Firmware-specific initialisation, checks, monitoring and control have to be performed. Generally, this is accomplished by writing custom scripts which utilize the underlying protocol and board wrappers. These scripts are then usually directly integrated into the set of scripts used during the operational lifetime of the instrument. These scripts are disjoint, can be written by different people, and have no consistency, unless this is explicitly enforced. The latter constraint is of particular concern when designing a general monitoring and control framework with multiple states, modes and status check routines.

To address this issue, pyfabil includes a firmware block `Plugin` mechanism. When a developer writes a new firmware block (where this block can in effect be the entire design), a plugin associated with this firmware must be implemented. This plugin must subclass the FirmwareBlock abstract class, define a number of parameters (implemented as decorators), implement the abstract methods defined in the superclass as well as any custom functionality. These plugins can then be dynamically loaded to any board instance during

runtime. The abstract methods make sure that all plugins provide sufficient functionality to be able to perform basic tests, initialise the firmware, routinely check the status of the running firmware and perform tests and diagnostics (assuming that these were properly implemented). The custom functionality implemented in these plugins can also be directly called. The list of available plugins is dynamically checked by the python wrapper, and when loading a plugin, it checks whether it is compatible with the board to which it is being loaded. It is up to the plugin developer to make sure that the loaded firmware on the board is correct (for example, by checking firmware name, version, and so on). It is generally desirable to be able to configure these plugins automatically. For example, during initialisation, a number of registers need to be set and checked for a particular observation. This can either be performed explicitly by the user, or by providing a configuration file which performs this automatically. Plugins will be covered in greater detail in section XXX.

The Python wrapper, together with the FPGA Board Access Layer, can either be used as a standalone tool, or in combination with other tools in more complex systems. In both cases, a standard and automatic way of configuring all the boards and firmware forming part of a deployment is desirable. A deployment of multiple boards and firmware will be referred to as an instrument. An instrument is composed of multiple boards, which can be of different types, each of which can have a number of firmware loaded (depending on the number of FPGAs on the board). Each board can have a different role in the setup. Instead of having to manually configure all boards and firmware, an `Instrument` management class is provided which, given a configuration file, will:

- Parse the configuration file
- Create all board instance, which can be of different types
- Connect to these boards
- Load the respective firmware on each board
- Load required plugins on each board
- Call the initialization routine on each board
- Call the initialization routine for each plugin
- Exposes functions for checking the status of the entire instrument, as well a drill-down view of the status of each board and plugin

The configuration file takes the form of an XML file. Since this feature is still in development, additional information will be provided in a later version of this document.

2 Generic FPGA Board

The main python package is called `pyfabil`. It contains a number of sub-packages:

base Implementations of base functionality, including the wrapper to the C++ shared library, the definitions file and a set of utilities

boards Implementations of FPGABoard class as well as all digital board sub classes

instruments Contains the generic instrument script and a placeholder for any custom instrument scripts

plugins Contains a sub-package for each board type, each having a set of plugins implemented for the particular board

tests Location of any unit tests and test scripts

The `FPGABoard` implements the basic functionality for interfacing with a digital board. The following list defines the functions available in this class, together with a brief descriptions of how they would be used. Since an instance of the this class cannot be created, an instance of the `TPM` will be used in the examples below. To following snippet can be used to instantiate a TPM instance:

```
from pyfabil import TPM

# Instantiate TPM instance
tpm = TPM()

# Instantiate TPM instance and connect to board
tpm = TPM(ip="127.0.0.1", port=10000)
```

The first statement imports the `TPM` class from `pyfabil` (Note that even though the TPM class is defined in `pyfabil.boards.tpm`, helper imports are defined in the __init__ file of the root package). The next statement then create a new instance of class (TPM). When an IP and port are specified, as in the third statement, the `connect` method in the library is called, setting up all the required internal data structures. It will also check whether it can communicate with the destination IP and port (in this case using the UCP protocol). A TPM simulator is also provided in `pyfabil.tests.tpm_simulator` in cases where a TPM is unavailable. This can be run in a separate console, and it will bind itself to all local IP addresses and wait for read and write requests. `FPGABoard` will keep an internal status for the device and all FPGAs defined for the board. If no error occurs, the status for a connected board will be `Status.OK`, otherwise if a network error occurs `Status.NetworkError`. The constructor will also set up logging and get a list of all available plugins which can be loaded for the board instance.

The status of the digital board upon connection depends on the board type as well as whether it (FPGAs on the board) has been programmed or not. Generally, if a firmware is already programmed the instance will get the query the register list and populate internal structures. These checks and the population of the register list is board-specific, so this functionality is implemented in the board classes. Generally, when connecting to a board which has already been programmed, the register is automatically generated and the user can then interact with them.

The programmed firmware can be changed by using the `load_firmware` method, as per the following snippet. This will re-program the specified device and re-load the register list. Note that the behavior of the command is board-specific, so please consult the appropriate sections in this document.

```
from pyfabil import Device
tpm.load_firmware(Device.FPGA_1, firmware="custom_design")
```

The first argument defines on which FPGA on the board the firmware will be loaded. The TPM has two FPGAs, and this statement will load the firmware labelled “custom_design”

on FPGA 1 (or ID 0). This call will 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. An additional call, `get_firmware_list` can be used to list the firmware available on the board (generally on Flash or over an NFS-mounted directory). Additional functions are also provided, defined in section XXX.

Once the board is connected and the register list is loaded, operations on registers, devices and memory block can be performed. Three types of operations can be performed:

- Read from and write to a memory address
- Read from and write to a register
- Read from and write to a memory address on a device (such as an SPI device)

A read and write method for each operation type is provided. Indexed access to the board instance can also be used to indirectly call these methods. Use indexed access results in less clutter in the code and makes it more readable. However, indexed access cannot be used in all cases, for example if a number of words or an offset needs to be provided, then the methods have to be called directly. The following snippet demonstrates these for a register (or memory block). The two sets of statements here are equivalent.

```
tpm.write_register('fpga1.register_name', 0x25)
val = tpm.read_register('fpga1.register_name', n = 1)

tpm['fpga1.register_name'] = 0x25
val = tpm['fpga1.register_name']
```

The first argument is the register name on which the operation will be performed. The structure of this name is board-specific. Generally, the device name with which the register is associated will be pre-pended to the actual name, such that identical register names on different FPGAs can be identified. The list of register can be grouped into components (as is the case for the TPM). The full register name must be match the internal representation of the register (as stored in the register dictionary). The list of registers can be acquired by calling `list_register_names`. Additional information for each register can also be analyzed by calling `get_register_list`. A user-friendly print out of the register list can be displayed by the statement:

```
print tpm
```

which will generated an output similar to:

```
 Device   Register                          Address      Bitmask
------   --------                          -------      -------
FPGA 1   regfile.block2048b                0x1000L      0xFFFFFFFF
FPGA 1   regfile.date_code                 0x0L         0xFFFFFFFF
...
Board    regfile.ada_ctrl.ada_a_ada4961    0x30000010L  0x00000008
Board    regfile.ada_ctrl.ada_fa_ada4961   0x30000010L  0x00000008
... (redacted)
```

```
# 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 first 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 requests 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 statements 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 specifies 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 defined 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()
```

3 iTPM

3.1 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

4 UniBoard

5 ROACH

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.