Python productivity for Zynq (Pynq) Documentation

Release 1.01

Xilinx

February 17, 2017

Contents

CHAPTER 1

What's New

Table of Contents

v1.4 PYNQ image

The latest PYNQ image for the Pynq-Z1 is available here:

- Download the v1.4 PYNQ-Z1 2017_02_10 image

This image corresponds to the v1.4 tag branch on the Pynq GitHub repository.

This version of the documentation refers to the new image. The previous version of the documentation, corresponding to the previous image release, can be accessed from the ReadTheDocs version menu.

In almost all cases, Python and IOP code developed for previous Pynq versions should be forwards compatible with the current image.

Summary of updates

- The Xilinx Linux kernel has been upgraded to 4.6.0.
- Python has been updated to 3.6.
- The Jupyter Notebook Extension has been added.
- The base overlay has been updated to include an interrupt controller from the IOPs to the Zynq PS.
- The IOPs have a direct interface to DRAM.
- The IOPs in the base overlay have an interrupt controller connected to the local peripherals in the IOP.
- The `asyncio` python package has been added and can be used to manage interrupts from an overlay
- The GPIO of the Arduino IOP has been modified
- The tracebuffer, video and PL APIs have been improved.
- Additional Pmod drivers added
- Additional Linux drivers added

- Linux library updates, and other bug fixes.

For a more complete list of changes, see the Pynq changelog

CHAPTER 2

Getting Started

Table of Contents

This guide will show you how to setup your computer and PYNQ-Z1 board to get started using PYNQ. Any issues can be posted to the PYNQ support forum.

Video Guide

You can watch the getting started video guide, or follow the instructions below.

Prerequisites

- PYNQ-Z1 board
- Computer with compatible browser (Supported Browsers)
- Ethernet cable
- Micro USB cable
- Micro-SD card with preloaded image, or blank card (Minimum 8GB recommended)

Get the image and prepare the Micro-SD Card

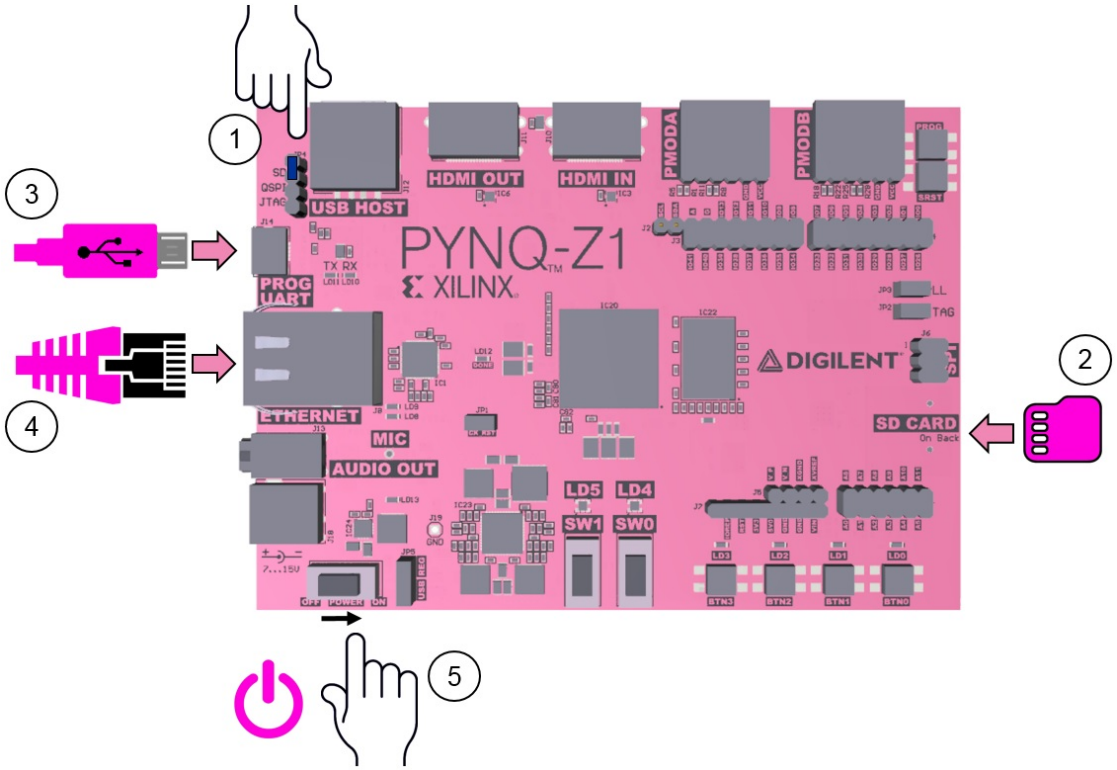
Preloaded Micro SD cards are available from Digilent. If you already have a Micro SD card preloaded with the PYNQ-Z1 image, you can skip this step.

To make your own PYNQ Micro-SD card:

- Download and the PYNQ-Z1 image and unzip
- **Write the image to a blank Micro SD card (minimum 8GB recommended)**
  - Windows: Use win32DiskImager
  - Linux/MacOS: Use the built in *dd* command*

* For detailed instructions for writing the SD card using different operating systems, see the Appendix: Writing the SD card image.

Setup the PYNQ-Z1



1. Set the *boot* jumper (labelled JP4 on the board) to the **SD** position by placing the jumper over the top two pins of JP4 as shown in the image. (This sets the board to boot from the Micro-SD card)
2. To power the PYNQ-Z1 from the micro USB cable, set the *power* jumper (JP5) to the **USB** position by placing the jumper over the top two pins of JP5 as shown in the image. (You can also power the board from an external 12V power regulator by setting the jumper to **REG**.)
3. Insert the **Micro SD** card loaded with the PYNQ-Z1 image into the board. (The Micro SD slot is underneath the board)
4. Connect the USB cable to your PC/Laptop, and to the **PROG/UART** (J14) on the board
5. Connect the Ethernet cable into your board and see the step below for connecting to a computer or network
6. The last step is to power on the board. You should follow the steps below to connect to the board before powering on.

Ethernet connection to the board

You can connect the Ethernet port of the PYNQ-Z1 Ethernet in the following ways:

- To a router or switch on the same network as your computer
- Directly to an Ethernet port on your computer

If available, you should connect your board to a network with Ethernet access. This will allow you to update your board and install new packages.

Connect to a network

If you connect to a network with a DHCP server, your board will automatically get an IP address. You must make sure you have permission to connect a device to your network, otherwise the board may not connect properly.

Router/Network switch (DHCP)
1. Connect to Ethernet port on router/switch
2. Browse to http://pynq:9090
3. Optional: Change hostname
4. Optional: Configure proxy

Hostname

The default hostname is `pynq`. If there is another device on the network with this hostname, you will need to change the hostname of your board **before** you connect it to the network. If you are not sure if there are other boards on the network, you should check if the *pynq* hostname is already in use before connecting a new board. One way to check this is by pinging *pynq* from a command prompt:

```
ping pynq
```

If you get a response from ping, this means there is already another device on the network with this hostname.

You can use a USB terminal connection to change the hostname **before** you connect your board to the network. If you are using a shared network, you should change the default hostname of the board in case other boards are connected to the network later.

You can also use the terminal to configure proxy settings, or to configure any other board settings. See below for detail on how to connect a terminal.

Connect directly to your computer

You will need to have an Ethernet port available on your computer, and you will need to have permissions to configure your network interface. With a direct connection, you will be able to use PYNQ, but unless you can bridge the Ethernet connection to the board to an Internet connection on your computer, your board will not have Internet access. You will be unable to update or load new packages without Internet access.

Direct Connection to your computer (Static IP)
1. Configure your computer with a Static IP*
2. Connect directly to your computer's Ethernet port
3. Browse to http://192.168.2.99:9090

* See Appendix: Assign your PC/Laptop a static IP address

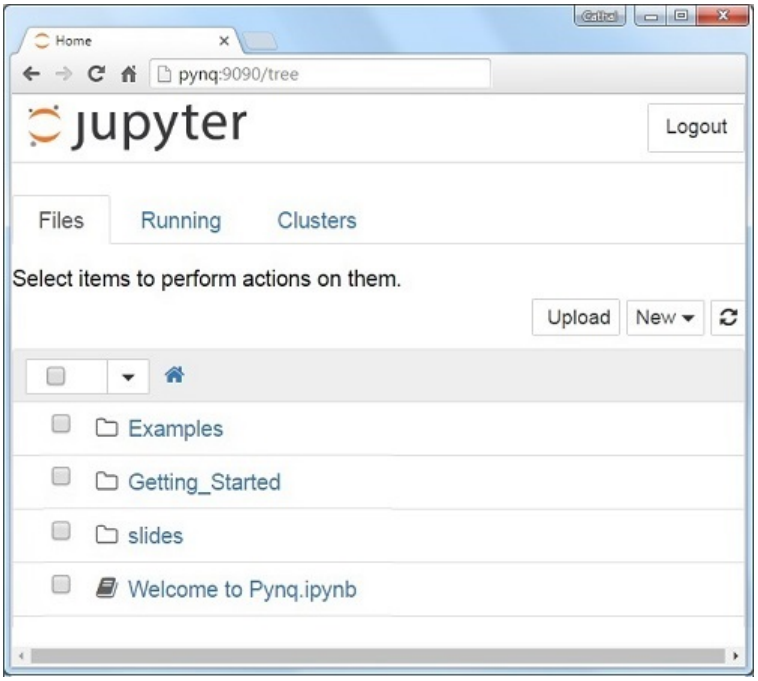
Powering on

As indicated in step 6 in the diagram above, slide the power switch to the *ON* position to *Turn On* the board. A *Red LED* will come on immediately to confirm that the board is powered on. After a few seconds, a *Yellow/Green LED* (LD12/DONE) will light up to show that the Zynq® device is operational.

After about 30 seconds you should see two blue LEDs and four yellow/green flash simultaneously. The blue LEDS will then go off while the yellow/green LEDS remain on. At this point the system is now booted and ready for use.

Connect to Jupyter

- Open a web browser and go to http://pynq:9090 (network) http://192.168.2.99:9090 (direct connection)
- The Jupyter username is xilinx and the password is also xilinx



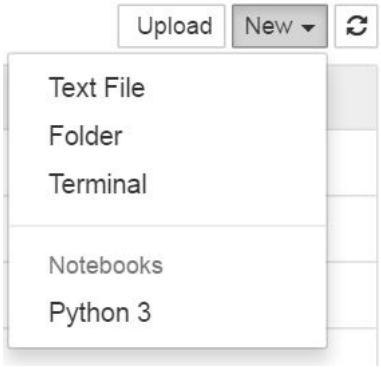
The default hostname is **pynq** and the default static IP address is `192.168.2.99`. If you changed the hostname or static IP of the board, you will need to change the address you browse to.

The first time you connect, it may take a few seconds for your computer to resolve the hostname/IP address.

Change hostname

If you are on a network where other pynq boards may be connected, you should change your hostname immediately. This is a common requirement in a work or university environment.

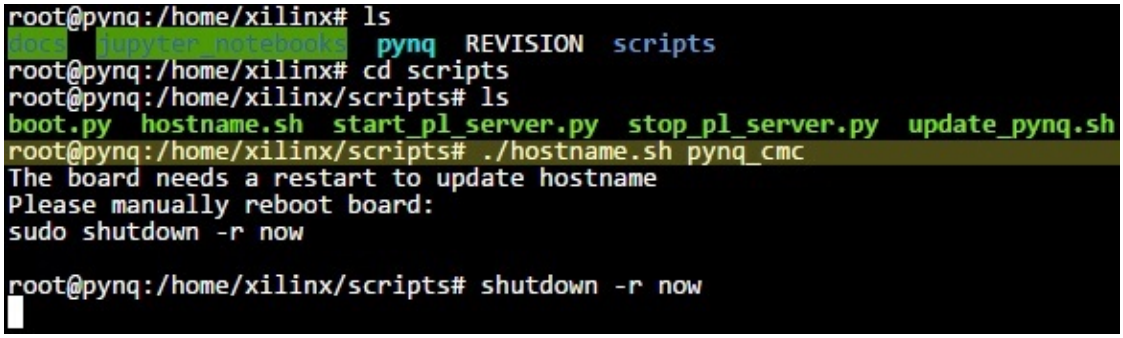
A terminal is available inside Jupyter. In the Jupyter portal home area, select **New >> terminal**.



This will open a terminal inside the browser as root.

Next enter and execute the following command. (Note that you should replace NEW_HOST_NAME with the hostname you want for your board.)

```
sudo /home/xilinx/scripts/hostname.sh NEW_HOST_NAME
```



Follow the instructions to reboot the board. Note that as you are logged in as root, sudo is not required, but if you are logged in as Xilinx, sudo must be added to these commands.

```
sudo shutdown -r now
```

When the board reboots, reconnect using the new hostname.

If you can't connect to your board, see the step below to open a terminal using the micro USB cable.

Connect to the PYNQ-Z1 board with a terminal connection over USB

If you need to change settings on the board but you can't access the terminal from Jupyter, you can connect a terminal over the micro USB cable that is already connected to the board. You can also use this terminal to check the network connection of the board. You will need to have terminal emulator software installed on your computer. PuTTY is available for free on Windows. To open a terminal, you will need to know the COM port for the board.

On Windows, you can find this in the Windows *Device Manager* in the control panel.

- Open the Device Manager, expand *Ports*

- Find the COM port for the *USB Serial Port*. e.g. COM5

Once you have the COM port, open PuTTY and use the following settings:

- Select serial
- Enter the COM port number
- Enter the baud rate
- Click *Open*

Hit *Enter* in the terminal window to make sure you can see the command prompt:

```
xilinnx@pynq:/home/xilinx#
```

Full terminal Settings:

- 115200 baud
- 8 data bits
- 1 stop bit
- No Parity
- No Flow Control

You can then run the same commands listed above to change the hostname, or configure a proxy.

You can also check the hostname of the board by running the *hostname* command:

```
hostname
```

You can also check the IP address of the board using *ifconfig*:

```
ifconfig
```

Configure proxy

If your board is connected to a network that uses a proxy, you need to set the proxy variables on the board. Open a terminal as above and enter the following where you should replace "my_http_proxy:8080" and "my_https_proxy:8080" with your settings.

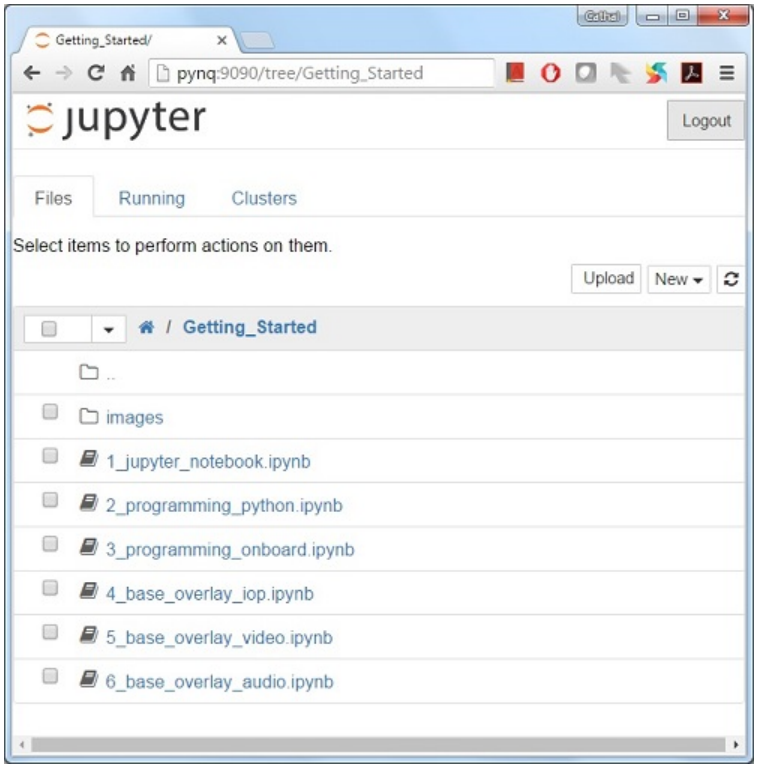
```
set http_proxy=my_http_proxy:8080
set https_proxy=my_https_proxy:8080
```

Using PYNQ

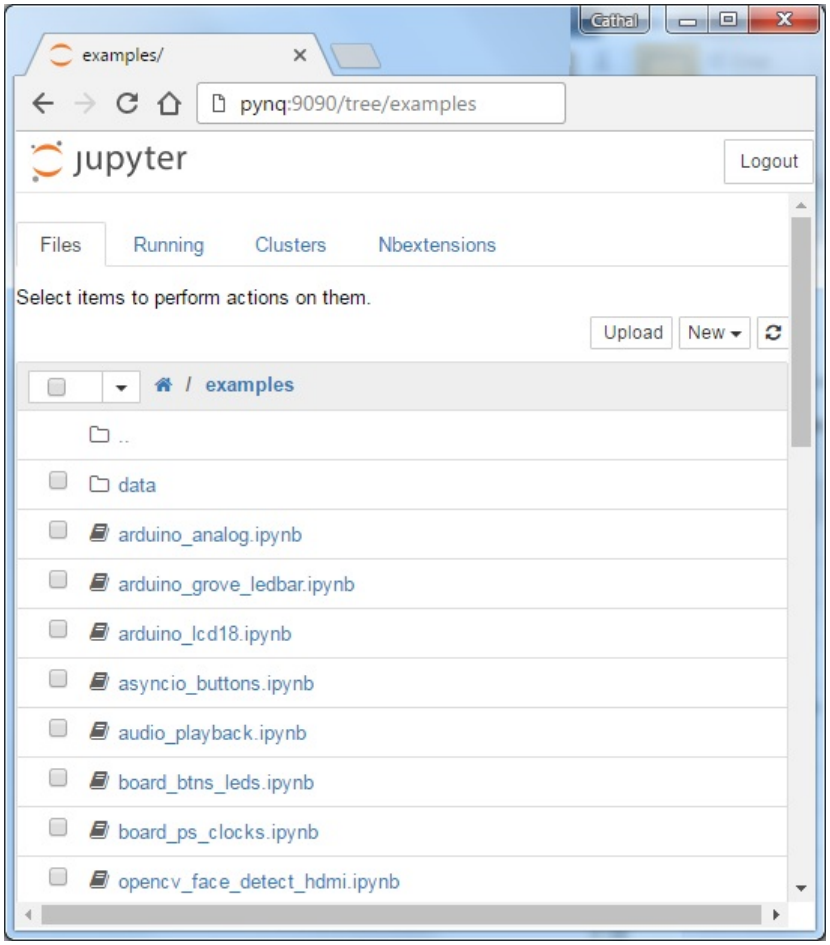
Getting started notebooks

A Jupyter notebook can be saved as html webpages. Some of this documentation has been generated directly from Jupyter notebooks.

You can view the documentation as a webpage, or if you have a board running PYNQ, you can view and run the notebook documentation interactively. The documentation available as notebooks can be found in the *Getting_Started* folder in the Jupyter home area.



There are also a number of example notebooks available showing how to use various peripherals with the board.



When you open a notebook and make any changes, or execute cells, the notebook document will be modified. It is recommended that you "Save a copy" when you open a new notebook. If you want to restore the original versions, you can download all the example notebooks from the PYNQ GitHub page .

Accessing files on the board

Samba, a file sharing service, is running on the board. The home area on the board can be accessed as a network drive, and you can transfer files to and from the board.

In Windows, to access the PYNQ home area you can go to:

```
\\pynq\xilinx
```

or

```
\\192.168.2.99\xilinx
```

Or in Linux:

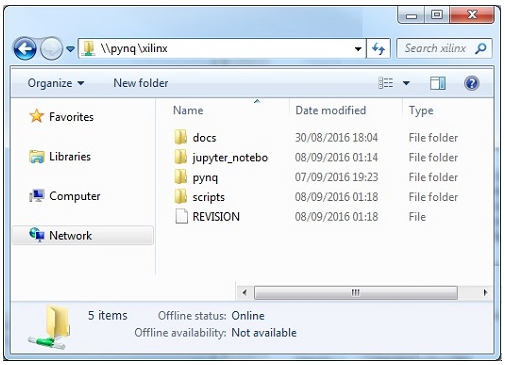
```
smb://pynq/xilinx
```

or

```
smb://192.168.2.99/xilinx
```

Remember to change the hostname/IP address if necessary.

The Samba username:password is `xilinx:xilinx`



Update PYNQ

You can update the pynq package by executing the script:

```
/home/xilinx/scripts/update_pynq.sh
```

This will check the pynq GitHub, download and install the latest release. Your board will need to have internet access to do this. Check the *Connect to a network* section above if necessary.

Updating PYNQ will overwrite the introductory and example notebooks. You should make sure you take a backup of this, and any code you added to the pynq python directory.

Troubleshooting

If you are having problems, please see the Frequently asked questions or go the PYNQ support forum

CHAPTER 3

PYNQ Introduction

Table of Contents

Project Goals

Xilinx® makes Zynq® devices, a class of All Programmable Systems on Chip (APSoC) which integrates a multi-core processor (Dual-core ARM® Cortex®-A9) and a Field Programmable Gate Array (FPGA) into a single integrated circuit. FPGA, or programmable logic, and microprocessors are complementary technologies for embedded systems. Each meets distinct requirements for embedded systems that the other cannot perform as well.

The main goal of **PYNQ**, **Py**thon Productivity for Zy**nq**, is to make it easier for designers of embedded systems to exploit the unique benefits of APSoCs in their applications. Specifically, PYNQ enables architects, engineers and programmers who design embedded systems to use Zynq APSoCs, without having to use ASIC-style design tools to design programmable logic circuits.

PYNQ achieves this goal in three main ways:

- Programmable logic circuits are presented as hardware libraries called *overlays*. These overlays are analogous to software libraries. A software engineer can select the overlay that best matches their application. The overlay can be accessed through an application programming interface (API). Creating a new overlay still requires engineers with expertise in designing programmable logic circuits. The key difference however, is the *build once, re-use many times* paradigm. Overlays, like software libraries, are designed to be configurable and re-used as often as possible in many different applications.

Note: This is a familiar approach that borrows from best-practice in the software community. Every day, the Linux kernel is used by hundreds of thousands of embedded designers. The kernel is developed and maintained by fewer than one thousand, high-skilled, software architects and engineers. The extensive re-use of the work of a relatively small number of very talented engineers enables many more software engineers to work at higher levels of abstraction. Hardware libraries or *overlays* are inspired by the success of the Linux kernel model in abstracting so many of the details of low-level, hardware-dependent software.

- PYNQ uses Python for programming both the embedded processors and the overlays. Python is a "productivity-level" language. To date, C or C++ are the most common, embedded programming languages. In contrast,

Python raises the level of programming abstraction and programmer productivity. These are not mutually-exclusive choices, however. PYNQ uses CPython which is written in C, and integrates thousands of C libraries and can be extended with optimized code written in C. Wherever practical, the more productive Python environment should be used, and whenever efficiency dictates, lower-level C code can be used.

- PYNQ is an open-source project that aims to work on any computing platform and operating system. This goal is achieved by adopting a web-based architecture, which is also browser agnostic. We incorporate the open-source Jupyter notebook infrastructure to run an Interactive Python (IPython) kernel and a web server directly on the ARM Cortex A9 of the Zynq device. The web server brokers access to the kernel via a suite of browser-based tools that provide a dashboard, bash terminal, code editors and Jupyter notebooks. The browser tools are implemented with a combination of JavaScript, HTML and CSS and run on any modern browser.

Summary

PYNQ is the first project to combine the following elements to simplify and improve APSoC design:

1. A high-level productivity language (Python in this case)
2. FPGA overlays with extensive APIs exposed as Python libraries
3. A web-based architecture served from the embedded processors, and
4. The Jupyter Notebook framework deployed in an embedded context

CHAPTER 4

Jupyter Notebook Introduction

Acknowledgements

The material in this tutorial is specific to Pynq. Wherever possible, however, it re-uses generic documentation describing Jupyter notebooks. In particular, we have re-used content from the following example notebooks:

1. What is the Jupyter Notebook?
2. Notebook Basics
3. Running Code
4. Markdown Cells

The original notebooks and further example notebooks are available at Jupyter documentation

Introduction

If you are reading this documentation from the webpage, you should note that the webpage is a static html version of the notebook from which it was generated. If the Pynq platform is available, you can open this notebook from the Getting_Started folder in the Pynq portal.

The Jupyter Notebook is an **interactive computing environment** that enables users to author notebook documents that include: - Live code - Interactive widgets - Plots - Narrative text - Equations - Images - Video

These documents provide a **complete and self-contained record of a computation** that can be converted to various formats and shared with others using email, Dropbox, version control systems (like git/GitHub) or nbviewer.jupyter.org.

Components

The Jupyter Notebook combines three components:

- **The notebook web application**: An interactive web application for writing and running code interactively and authoring notebook documents.
- **Kernels**: Separate processes started by the notebook web application that runs users' code in a given language and returns output back to the notebook web application. The kernel also handles things like computations for interactive widgets, tab completion and introspection.

- **Notebook documents**: Self-contained documents that contain a representation of all content in the notebook web application, including inputs and outputs of the computations, narrative text, equations, images, and rich media representations of objects. Each notebook document has its own kernel.

Notebook web application

The notebook web application enables users to:

- **Edit code in the browser**, with automatic syntax highlighting, indentation, and tab completion/introspection.
- **Run code from the browser**, with the results of computations attached to the code which generated them.
- See the results of computations with **rich media representations**, such as HTML, LaTeX, PNG, SVG, PDF, etc.
- Create and use **interactive JavaScript widgets**, which bind interactive user interface controls and visualizations to reactive kernel side computations.
- Author **narrative text** using the Markdown markup language.
- Build **hierarchical documents** that are organized into sections with different levels of headings.
- Include mathematical equations using **LaTeX syntax in Markdown**, which are rendered in-browser by MathJax.

Kernels

The Notebook supports a range of different programming languages. For each notebook that a user opens, the web application starts a kernel that runs the code for that notebook. Each kernel is capable of running code in a single programming language. There are kernels available in the following languages:

- Python(https://github.com/ipython/ipython)
- Julia (https://github.com/JuliaLang/IJulia.jl)
- R (https://github.com/takluyver/IRkernel)
- Ruby (https://github.com/minrk/iruby)
- Haskell (https://github.com/gibiansky/IHaskell)
- Scala (https://github.com/Bridgewater/scala-notebook)
- node.js (https://gist.github.com/Carreau/4279371)
- Go (https://github.com/takluyver/igo)

The default kernel runs Python code which is the language Pynq is based on.

Kernels communicate with the notebook web application and web browser using a JSON over ZeroMQ/WebSockets message protocol that is described here. Most users don't need to know about these details, but it helps to understand that "kernels run code."

Notebook documents

Notebook documents contain the **inputs and outputs** of an interactive session as well as **narrative text** that accompanies the code but is not meant for execution. **Rich output** generated by running code, including HTML, images, video, and plots, is embedded in the notebook, which makes it a complete and self-contained record of a computation.

When you run the notebook web application on your computer, notebook documents are just **files** on your local filesystem with a **‘‘.ipynb‘‘** extension. This allows you to use familiar workflows for organizing your notebooks into folders and sharing them with others.

Notebooks consist of a **linear sequence of cells**. There are four basic cell types:

- **Code cells:** Input and output of live code that is run in the kernel
- **Markdown cells:** Narrative text with embedded LaTeX equations
- **Heading cells:** Deprecated. Headings are supported in Markdown cells
- **Raw cells:** Unformatted text that is included, without modification, when notebooks are converted to different formats using nbconvert

Internally, notebook documents are JSON data with binary values base64 encoded. This allows them to be **read and manipulated programmatically** by any programming language. Because JSON is a text format, notebook documents are version control friendly.

Notebooks can be exported to different static formats including HTML, reStructeredText, LaTeX, PDF, and slide shows (reveal.js) using Jupyter's `nbconvert` utility. Some of documentation for Pynq, including this page, was written in a Notebook and converted to html for hosting on the project's documentation website.

Furthermore, any notebook document available from a **public URL or on GitHub can be shared** via nbviewer. This service loads the notebook document from the URL and renders it as a static web page. The resulting web page may thus be shared with others **without their needing to install the Jupyter Notebook**.

GitHub also renders notebooks, so any Notebook added to GitHub can be viewed as intended.

Notebook Basics

The Notebook dashboard

The Notebook server runs on the ARM® processor of the PYNQ-Z1. You can open the notebook dashboard by navigating to pynq:9090 when your board is connected to the network. The dashboard serves as a home page for notebooks. Its main purpose is to display the notebooks and files in the current directory. For example, here is a screenshot of the dashboard page for the `Examples` directory in the Jupyter repository:

The top of the notebook list displays clickable breadcrumbs of the current directory. By clicking on these breadcrumbs or on sub-directories in the notebook list, you can navigate your filesystem.

To create a new notebook, click on the “New” button at the top of the list and select a kernel from the dropdown (as seen below).

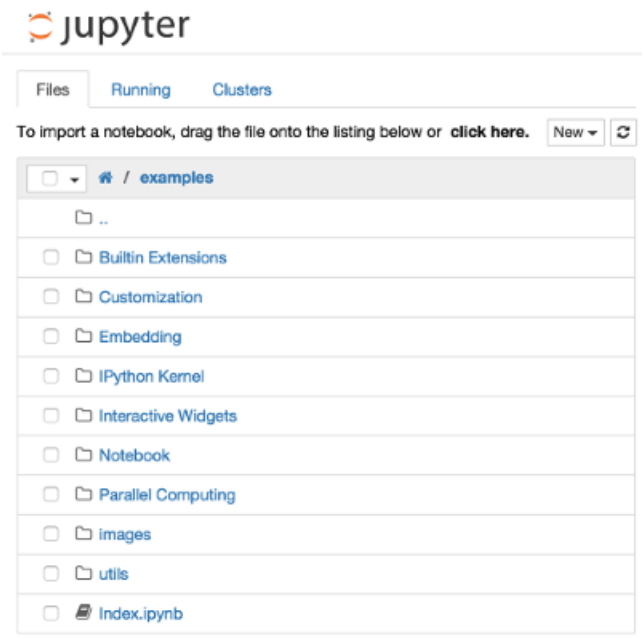
Notebooks and files can be uploaded to the current directory by dragging a notebook file onto the notebook list or by the “click here” text above the list.

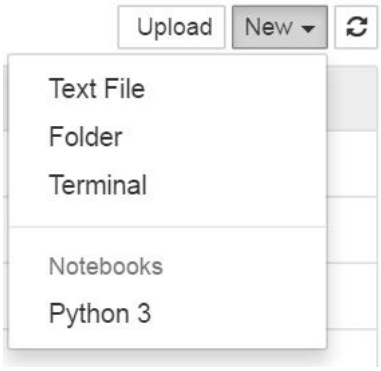
The notebook list shows green “Running” text and a green notebook icon next to running notebooks (as seen below). Notebooks remain running until you explicitly shut them down; closing the notebook's page is not sufficient.

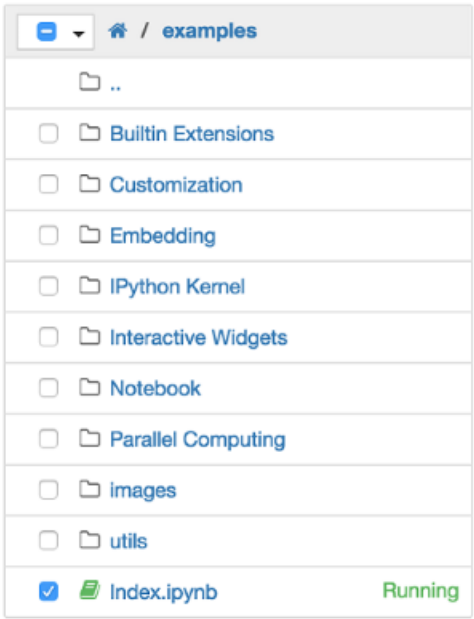
To shutdown, delete, duplicate, or rename a notebook check the checkbox next to it and an array of controls will appear at the top of the notebook list (as seen below). You can also use the same operations on directories and files when applicable.

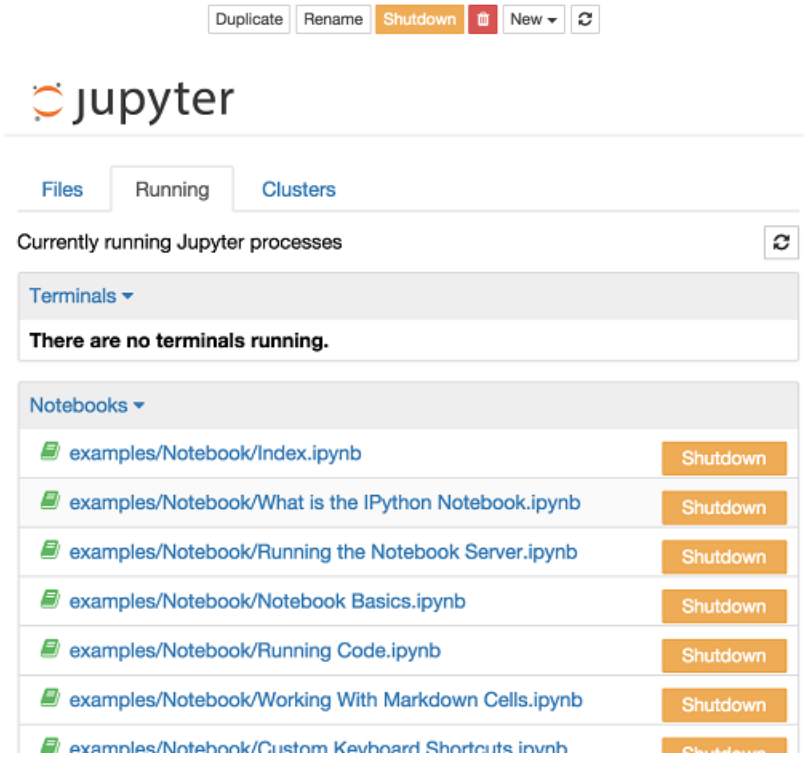
To see all of your running notebooks along with their directories, click on the “Running” tab:

This view provides a convenient way to track notebooks that you start as you navigate the filesystem in a long running notebook server.


jupyter
Files
Running
Clusters
To import a notebook, drag the file onto the listing below or click here.
New
/ examples
..
Builtin Extensions
Customization
Embedding
IPython Kernel
Interactive Widgets
Notebook
Parallel Computing
images
utils
Index.ipynb


Upload
New
Text File
Folder
Terminal
Notebooks
Python 3


/ examples
..
Builtin Extensions
Customization
Embedding
IPython Kernel
Interactive Widgets
Notebook
Parallel Computing
images
utils
Index.ipynb
Running



Overview of the Notebook UI

If you create a new notebook or open an existing one, you will be taken to the notebook user interface (UI). This UI allows you to run code and author notebook documents interactively. The notebook UI has the following main areas:

- Menu
- Toolbar
- Notebook area and cells

The notebook has an interactive tour of these elements that can be started in the "Help:User Interface Tour" menu item.

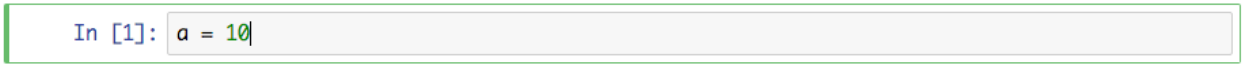
Modal editor

The Jupyter Notebook has a modal user interface which means that the keyboard does different things depending on which mode the Notebook is in. There are two modes: edit mode and command mode.

Edit mode

Edit mode is indicated by a green cell border and a prompt showing in the editor area:

When a cell is in edit mode, you can type into the cell, like a normal text editor.



Enter edit mode by pressing `Enter` or using the mouse to click on a cell's editor area.

Command mode

Command mode is indicated by a grey cell border with a blue left margin:

When you are in command mode, you are able to edit the notebook as a whole, but not type into individual cells. Most importantly, in command mode, the keyboard is mapped to a set of shortcuts that let you perform notebook and cell actions efficiently. For example, if you are in command mode and you press `c`, you will copy the current cell - no modifier is needed.

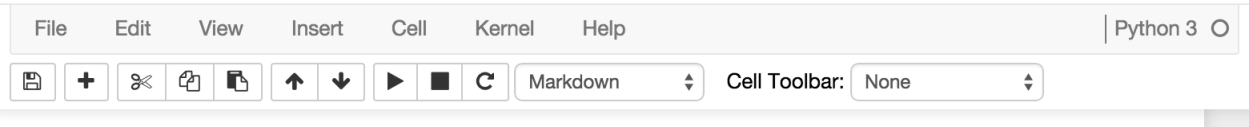
```
In [1]: a = 10
```

Don't try to type into a cell in command mode; unexpected things will happen!

Enter command mode by pressing `Esc` or using the mouse to click *outside* a cell's editor area.

Mouse navigation

All navigation and actions in the Notebook are available using the mouse through the menubar and toolbar, both of which are above the main Notebook area:



Cells can be selected by clicking on them with the mouse. The currently selected cell gets a grey or green border depending on whether the notebook is in edit or command mode. If you click inside a cell's editor area, you will enter edit mode. If you click on the prompt or output area of a cell you will enter command mode.

If you are running this notebook in a live session on the PYNQ-Z1, try selecting different cells and going between edit and command mode. Try typing into a cell.

If you want to run the code in a cell, you would select it and click the `play` button in the toolbar, the "Cell:Run" menu item, or type Ctrl + Enter. Similarly, to copy a cell you would select it and click the `copy` button in the toolbar or the "Edit:Copy" menu item. Ctrl + C, V are also supported.

Markdown and heading cells have one other state that can be modified with the mouse. These cells can either be rendered or unrendered. When they are rendered, you will see a nice formatted representation of the cell's contents. When they are unrendered, you will see the raw text source of the cell. To render the selected cell with the mouse, and execute it. (Click the `play` button in the toolbar or the "Cell:Run" menu item, or type Ctrl + Enter. To unrender the selected cell, double click on the cell.

Keyboard Navigation

There are two different sets of keyboard shortcuts: one set that is active in edit mode and another in command mode.

The most important keyboard shortcuts are `Enter`, which enters edit mode, and `Esc`, which enters command mode.

In edit mode, most of the keyboard is dedicated to typing into the cell's editor. Thus, in edit mode there are relatively few shortcuts. In command mode, the entire keyboard is available for shortcuts, so there are many more. The `Help->`"Keyboard Shortcuts" dialog lists the available shortcuts.

Some of the most useful shortcuts are:

1. Basic navigation: `enter`, `shift-enter`, `up/k`, `down/j`
2. Saving the notebook: `s`
3. Change Cell types: `y`, `m`, `1-6`, `t`
4. Cell creation: `a`, `b`
5. Cell editing: `x`, `c`, `v`, `d`, `z`
6. Kernel operations: `i`, `0` (press twice)

Running Code

First and foremost, the Jupyter Notebook is an interactive environment for writing and running code. The notebook is capable of running code in a wide range of languages. However, each notebook is associated with a single kernel. Pynq, and this notebook is associated with the IPython kernel, which runs Python code.

Code cells allow you to enter and run code

Run a code cell using `Shift-Enter` or pressing the `play` button in the toolbar above. The button displays *run cell, select below* when you hover over it.

```
In [1]: a = 10
```

```
In [ ]: print(a)
```

```
10
```

There are two other keyboard shortcuts for running code:

- `Alt-Enter` runs the current cell and inserts a new one below.
- `Ctrl-Enter` run the current cell and enters command mode.

Managing the Kernel

Code is run in a separate process called the Kernel. The Kernel can be interrupted or restarted. Try running the following cell and then hit the `stop` button in the toolbar above. The button displays *interrupt kernel* when you hover over it.

```
In [ ]: import time
        time.sleep(10)
```

Cell menu

The "Cell" menu has a number of menu items for running code in different ways. These includes:

- Run and Select Below
- Run and Insert Below
- Run All
- Run All Above
- Run All Below

Restarting the kernels

The kernel maintains the state of a notebook's computations. You can reset this state by restarting the kernel. This is done from the menu bar, or by clicking on the corresponding button in the toolbar.

sys.stdout

The stdout and stderr streams are displayed as text in the output area.

```
In [ ]: print("Hello from Pynq!")
```

Output is asynchronous

All output is displayed asynchronously as it is generated in the Kernel. If you execute the next cell, you will see the output one piece at a time, not all at the end.

```
In [ ]: import time, sys
        for i in range(8):
            print(i)
            time.sleep(0.5)
```

Large outputs

To better handle large outputs, the output area can be collapsed. Run the following cell and then single- or double-click on the active area to the left of the output:

```
In [ ]: for i in range(50):
            print(i)
```

Markdown

Text can be added to Jupyter Notebooks using Markdown cells. Markdown is a popular markup language that is a superset of HTML. Its specification can be found here:

http://daringfireball.net/projects/markdown/

Markdown basics

You can make text *italic* or **bold**.

You can build nested itemized or enumerated lists:

- One
  - Sublist
    - This
- Sublist - That - The other thing
- Two
- Sublist
- Three

- Sublist

Now another list:

1. Here we go
    (a) Sublist
    (b) Sublist
2. There we go
3. Now this

You can add horizontal rules:

Here is a blockquote:

> Beautiful is better than ugly. Explicit is better than implicit. Simple is better than complex. Complex is better than complicated. Flat is better than nested. Sparse is better than dense. Readability counts. Special cases aren't special enough to break the rules. Although practicality beats purity. Errors should never pass silently. Unless explicitly silenced. In the face of ambiguity, refuse the temptation to guess. There should be one– and preferably only one –obvious way to do it. Although that way may not be obvious at first unless you're Dutch. Now is better than never. Although never is often better than *right* now. If the implementation is hard to explain, it's a bad idea. If the implementation is easy to explain, it may be a good idea. Namespaces are one honking great idea – let's do more of those!

And shorthand for links:

Jupyter's website

Headings

You can add headings by starting a line with one (or multiple) # followed by a space, as in the following example:

```
# Heading 1
# Heading 2
## Heading 2.1
## Heading 2.2
```

Embedded code

You can embed code meant for illustration instead of execution in Python:

```
def f(x):
    """a docstring"""
    return x**2
```

or other languages:

```
if (i=0; i<n; i++) {
  printf("hello %d\n", i);
  x += 4;
}
```

LaTeX equations

Courtesy of MathJax, you can include mathematical expressions both inline: $e^{i\pi} + 1 = 0$ and displayed:

$$e^x = \sum_{i=0}^{\infty} \frac{1}{i!}x^i$$

$$Inline expressions can be added by surrounding the latex code with$$

$:

```
$e^{i\pi} + 1 = 0$
```

Expressions on their own line are surrounded by $$:

```
$$e^x=\sum_{i=0}^\infty \frac{1}{i!}x^i$$
```

GitHub flavored markdown

The Notebook webapp supports Github flavored markdown meaning that you can use triple backticks for code blocks:

```
<pre>
```python
print "Hello World"
```
</pre>

<pre>
```javascript
console.log("Hello World")
```
</pre>
```

Gives:

```
print "Hello World"
```

```
console.log("Hello World")
```

And a table like this:

```
<pre>
```

This	is
a	table

```
</pre>
```

A nice HTML Table:

This	is
a	table

General HTML

Because Markdown is a superset of HTML you can even add things like HTML tables:

Header 1

Header 2

row 1, cell 1

row 1, cell 2

row 2, cell 1

row 2, cell 2

Local files

If you have local files in your Notebook directory, you can refer to these files in Markdown cells directly:

```
[subdirectory/]<filename>
```

Security of local files

Note that this means that the Jupyter notebook server also acts as a generic file server for files inside the same tree as your notebooks. Access is not granted outside the notebook folder so you have strict control over what files are visible, but for this reason it is highly recommended that you do not run the notebook server with a notebook directory at a high level in your filesystem (e.g. your home directory).

When you run the notebook in a password-protected manner, local file access is restricted to authenticated users unless read-only views are active.

CHAPTER 5

Cortex-A9 programming in Python

We show here an example of how to run Python with Pynq. Python is running exclusively on the ARM Cortex-A9 processor. This example, which is based on calculating the factors and primes of integer numbers, give us a sense of the performance available when running on a 650MHz ARM Cortex-A9 dual core processor running Linux.

The factors and primes example

Code is provided in the cell below for a function to calculate factors and primes. It contains some sample functions to calculate the factors and primes of integers. We will use three functions from the factors_and_primes module to demonstrate Python programming.

In [1]:
```
"""Factors-and-primes functions.

    Find factors or primes of integers, int ranges and int lists
    and sets of integers with most factors in a given integer interval

    """

    from pprint import pprint

    def factorize(n):
        """Calculate all factors of integer n.

        Parameters
        ----------
        n : int
            integer to factorize.

        Returns
        -------
        list
            A sorted set of integer factors of n.

        """
        factors = []
        if isinstance(n, int) and n > 0:
            if n == 1:
                factors.append(n)
                return factors
```

```
            else:
                for x in range(1, int(n**0.5)+1):
                    if n % x == 0:
                        factors.append(x)
                        factors.append(n//x)
                return sorted(set(factors))
        else:
            print('factorize ONLY computes with one integer argument > 0')


    def primes_between(interval_min, interval_max):
        """Find all primes in the interval.

        The interval is defined by interval_min and interval_max.

        Parameters
        ----------
        interval_min : int
            Start of the integer range.
        interval_max : int
            End of the integer range.

        Returns
        -------
        list
            A sorted set of integer primes in original range.

        """
        primes = []
        if (isinstance(interval_min, int) and interval_min > 0 and
           isinstance(interval_max, int) and interval_max > interval_min):
            if interval_min == 1:
                primes = [1]
            for i in range(interval_min, interval_max):
                if len(factorize(i)) == 2:
                    primes.append(i)
            return sorted(primes)
        else:
            print('primes_between ONLY computes over the specified range.')


    def primes_in(integer_list):
        """Calculate all unique prime numbers.

        Calculate the prime numbers in a list of integers.

        Parameters
        ----------
        integer_list : list
            A list of integers to test for primality.

        Returns
        -------
        list
```

```
        A sorted set of integer primes from original list.

    """
    primes = []
    try:
        for i in (integer_list):
            if len(factorize(i)) == 2:
                primes.append(i)
        return sorted(set(primes))
    except TypeError:
        print('primes_in ONLY computes over lists of integers.')


def get_ints_with_most_factors(interval_min, interval_max):
    """Finds the integers with the most factors.

    Find the integer or integers with the most factors in a given
    integer range.

    The returned result is a list of tuples, where each tuple is:
    [no_with_most_factors (int), no_of_factors (int),
    factors (int list)].

    Parameters
    ----------
    interval_min : int
        Start of the integer range.
    interval_max : int
        End of the integer range.

    Returns
    -------
    list
        A list of tuples showing the results.

    """
    max_no_of_factors = 1
    all_ints_with_most_factors = []

    #: Find the lowest number with most factors between i_min and i_max
    if interval_check(interval_min, interval_max):
        for i in range(interval_min, interval_max):
            factors_of_i = factorize(i)
            no_of_factors = len(factors_of_i)
            if no_of_factors > max_no_of_factors:
                max_no_of_factors = no_of_factors
                results = (i, max_no_of_factors, factors_of_i,\
                           primes_in(factors_of_i))
        all_ints_with_most_factors.append(results)

        #: Find any larger numbers with an equal number of factors
        for i in range(all_ints_with_most_factors[0][0]+1, interval_max):
            factors_of_i = factorize(i)
            no_of_factors = len(factors_of_i)
```

```
            if no_of_factors == max_no_of_factors:
                results = (i, max_no_of_factors, factors_of_i, \
                           primes_in(factors_of_i))
                all_ints_with_most_factors.append(results)
        return all_ints_with_most_factors
    else:
        print_error_msg()


def print_ints_with_most_factors(interval_min, interval_max):
    """Reports integers with most factors in a given integer range.

    The results can consist of the following:
    1.  All the integers with the most factors
    2.  The number of factors
    3.  The actual factors of each of the integers
    4.  Any prime numbers in the list of factors

    Parameters
    ----------
    interval_min : int
        Start of the integer range.
    interval_max : int
        End of the integer range.

    Returns
    -------
    list
        A list of tuples showing the integers and factors.

    """
    if interval_check(interval_min, interval_max):
        print('\nBetween {} and {} the number/s with the most factors:\n'.
              format(interval_min, interval_max))
        for results in (get_ints_with_most_factors(
                           interval_min, interval_max)):
            print('{} ... with the following {} factors:'
                  .format(results[0], results[1]))
            pprint(results[2])
            print('The prime number factors of {} are:'
                  .format(results[0]))
            pprint(results[3])
    else:
        print_error_msg()


def interval_check(interval_min, interval_max):
    """Check type and range of integer interval.

    Parameters
    ----------
    interval_min : int
        Start of the integer range.
    interval_max : int
```

```
            End of the integer range.

        Returns
        -------
        None

        """
        if (isinstance(interval_min, int) and interval_min > 0 and
           isinstance(interval_max, int) and interval_max > interval_min):
            return True
        else:
            return False

    def print_error_msg():
        """Print invalid integer interval error message.

        Returns
        -------
        None

        """
        print('ints_with_most_factors ONLY computes over integer intervals where'
              ' interval_min <= int_with_most_factors < interval_max and'
              ' interval_min >= 1')
```

Next we will call the factorize() function to calculate the factors of an integer.

```
In [2]: factorize(1066)
```

```
Out[2]: [1, 2, 13, 26, 41, 82, 533, 1066]
```

The primes_between() function can tell us how many prime numbers there are in an integer range. Let's try it for the interval 1 through 1066. We can also use one of Python's built-in methods len() to count them all.

```
In [3]: len(primes_between(1, 1066))
```

```
Out[3]: 180
```

Additionally, we can combine len() with another built-in method, sum(), to calculate the average of the 180 prime numbers.

```
In [4]: primes_1066 = primes_between(1, 1066)
        primes_1066_average = sum(primes_1066) / len(primes_1066)
        primes_1066_average
```

```
Out[4]: 486.2055555555556
```

This result makes sense intuitively because prime numbers are known to become less frequent for larger number intervals. These examples demonstrate how Python treats functions as first-class objects so that functions may be passed as parameters to other functions. This is a key property of functional programming and demonstrates the power of Python.

In the next code snippet, we can use list comprehensions (a 'Pythonic' form of the map-filter-reduce template) to 'mine' the factors of 1066 to find those factors that end in the digit '3'.

```
In [5]: primes_1066_ends3 = [x for x in primes_between(1, 1066) if str(x).endswith('3')]
        primes_1066_ends3
```

```
Out[5]: [3,
         13,
```

```
23,
43,
53,
73,
83,
103,
113,
163,
173,
193,
223,
233,
263,
283,
293,
313,
353,
373,
383,
433,
443,
463,
503,
523,
563,
593,
613,
643,
653,
673,
683,
733,
743,
773,
823,
853,
863,
883,
953,
983,
1013,
1033,
1063]
```

This code tells Python to first convert each prime between 1 and 1066 to a string and then to return those numbers whose string representation end with the number '3'. It uses the built-in str() and endswith() methods to test each prime for inclusion in the list.

And because we really want to know what fraction of the 180 primes of 1066 end in a '3', we can calculate ...

```
In [6]: len(primes_1066_ends3) / len(primes_1066)
```

```
Out[6]: 0.25
```

These examples demonstrate how Python is a modern, multi-paradigmatic language. More simply, it continually integrates the best features of other leading languages, including functional programming constructs. Consider how

many lines of code you would need to implement the list comprehension above in C and you get an appreciation of the power of productivity-layer languages. Higher levels of programming abstraction really do result in higher programmer productivity!

More intensive calculations

To stress the ARM processor a little more, we will run a script to determine the integer number, or numbers, that have the most factors between 1 and 1066, using the print_ints_with_most_factors() function from the factors_and_primes module.

```
In [7]: print_ints_with_most_factors(1, 1066)


Between 1 and 1066 the number/s with the most factors:

840 ... with the following 32 factors:
[1,
 2,
 3,
 4,
 5,
 6,
 7,
 8,
 10,
 12,
 14,
 15,
 20,
 21,
 24,
 28,
 30,
 35,
 40,
 42,
 56,
 60,
 70,
 84,
 105,
 120,
 140,
 168,
 210,
 280,
 420,
 840]
The prime number factors of 840 are:
[2, 3, 5, 7]
```

The ARM processor remains quite responsive. Running this for much larger numbers, say 50,000, will demonstrate noticeably slower responses as we would expect.

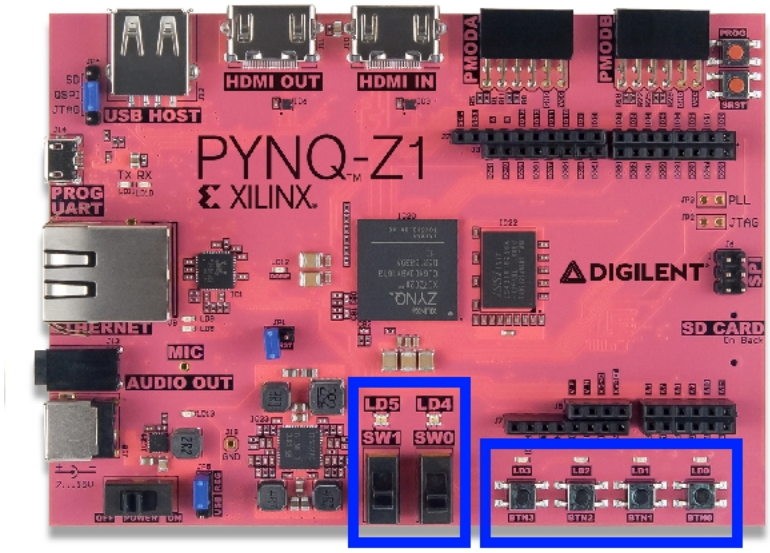
Programming PYNQ-Z1's onboard peripherals

LEDs, switches and buttons

PYNQ-Z1 has the following on-board LEDs, pushbuttons and switches:

- 4 monochrome LEDs (LD3-LD0)
- 4 push-button switches (BTN3-BTN0)
- 2 RGB LEDs (LD5-LD4)
- 2 Slide-switches (SW1-SW0)

The peripherals are highlighted in the image below.



All of these peripherals are connected to programmable logic. This means controllers must be implemented in an overlay before these peripherals can be used. The base overlay contains controllers for all of these peripherals.

Note that there are additional push-buttons and LEDs on the board (e.g. power LED, reset button). They are not user accessible, and are not highlighted in the figure.

Peripheral Example

Using the base overlay, each of the highlighted devices can be controlled using their corresponding pynq classes.

To demonstrate this, we will first download the base overlay to ensure it is loaded, and then import the LED, RGBLED, Switch and Button classes from the module `pynq.board`.

```
In [1]: from pynq import Overlay
        from pynq.board import LED
        from pynq.board import RGBLED
        from pynq.board import Switch
        from pynq.board import Button

        Overlay("base.bit").download()
```

Controlling a single LED

Now we can instantiate objects of each of these classes and use their methods to manipulate the corresponding peripherals. Let's start by instantiating a single LED and turning it on and off.

```
In [2]: led0 = LED(0)
```

```
In [3]: led0.on()
```

Check the board and confirm the LD0 is ON

```
In [4]: led0.off()
```

Let's then toggle *led0* using the sleep() method to see the LED flashing.

```
In [5]: import time
        from pynq.board import LED
        from pynq.board import Button

        led0 = LED(0)
        for i in range(20):
            led0.toggle()
            time.sleep(.1)
```

Example: Controlling all the LEDs, switches and buttons

The example below creates 3 separate lists, called leds, switches and buttons.

```
In [6]: MAX_LEDS = 4
        MAX_SWITCHES = 2
        MAX_BUTTONS = 4

        leds = [0] * MAX_LEDS
        switches = [0] * MAX_SWITCHES
        buttons = [0] * MAX_BUTTONS

        for i in range(MAX_LEDS):
            leds[i] = LED(i)
        for i in range(MAX_SWITCHES):
```

```
            switches[i] = Switch(i)
        for i in range(MAX_BUTTONS):
            buttons[i] = Button(i)
```

It will be useful to be able to turn off selected LEDs so we will create a helper function to do that. It either clears the LEDs whose numbers we list in the parameter, or by default clears LD3-LD0.

```
In [7]: # Helper function to clear LEDs
        def clear_LEDs(LED_nos=list(range(MAX_LEDS))):
            """Clear LEDS LD3-0 or the LEDs whose numbers appear in the list"""
            for i in LED_nos:
                leds[i].off()

        clear_LEDs()
```

First, all LEDs are set to off. Then each switch is read, and if in the on position, the corresponding led is turned on. You can execute this cell a few times, changing the position of the switches on the board.

- LEDs start in the off state
- If SW0 is on, LD2 and LD0 will be on
- If SW1 is on, LD3 and LD1 will be on

```
In [8]: clear_LEDs()

        for i in range(MAX_LEDS):
            if switches[i%2].read():
                leds[i].on()
            else:
                leds[i].off()
```

The last example toggles an led (on or off) if its corresponding push button is pressed for so long as SW0 is switched on.

To end the program, slide SW0 to the off position.

```
In [9]: import time

        clear_LEDs()

        while switches[0].read():
            for i in range(MAX_LEDS):
                if buttons[i].read():
                    leds[i].toggle()
                    time.sleep(.1)

        clear_LEDs()
```

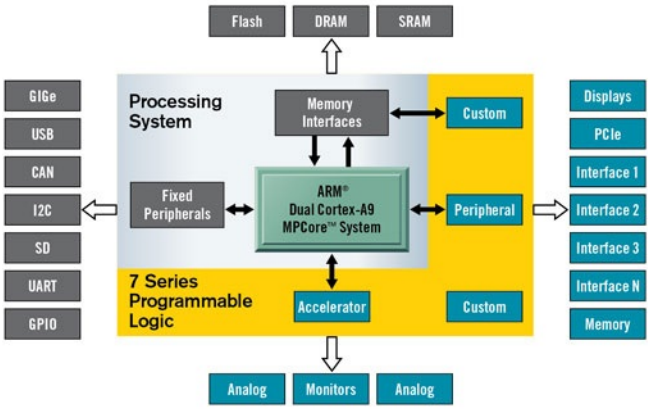
CHAPTER 7

Introduction to Overlays

Table of Contents

Overlay Concept

The Xilinx® Zynq® All Programmable device is an SOC based on a dual-core ARM® Cortex®-A9 processor (referred to as the *Processing System* or **PS**). A Zynq chip also includes FPGA fabric (referred to as *Programmable Logic* or **PL**). The ARM SoC subsystem also includes a number of dedicated peripherals (memory controllers, USB, Uart, IIC, SPI etc).



On the Pynq-Z1 board, the DDR memory controller, Ethernet, USB, SD, UART are connected on the board.

The FPGA fabric is reconfigurable, and can be used to implement high performance functions in hardware. However, FPGA design is a specialized task which requires deep hardware engineering knowledge and expertise. Overlays, or hardware libraries, are programmable/configurable FPGA designs that extend the user application from the Processing System of the Zynq into the Programmable Logic. Overlays can be used to accelerate a software application, or to customize the hardware platform for a particular application.

For example, image processing is a typical application where the FPGAs can provide acceleration. A software programmer can use an overlay in a similar way to a software library to run some of the image processing functions (e.g. edge detect, thresholding etc.) on the FPGA fabric. Overlays can be loaded to the FPGA dynamically, as required,

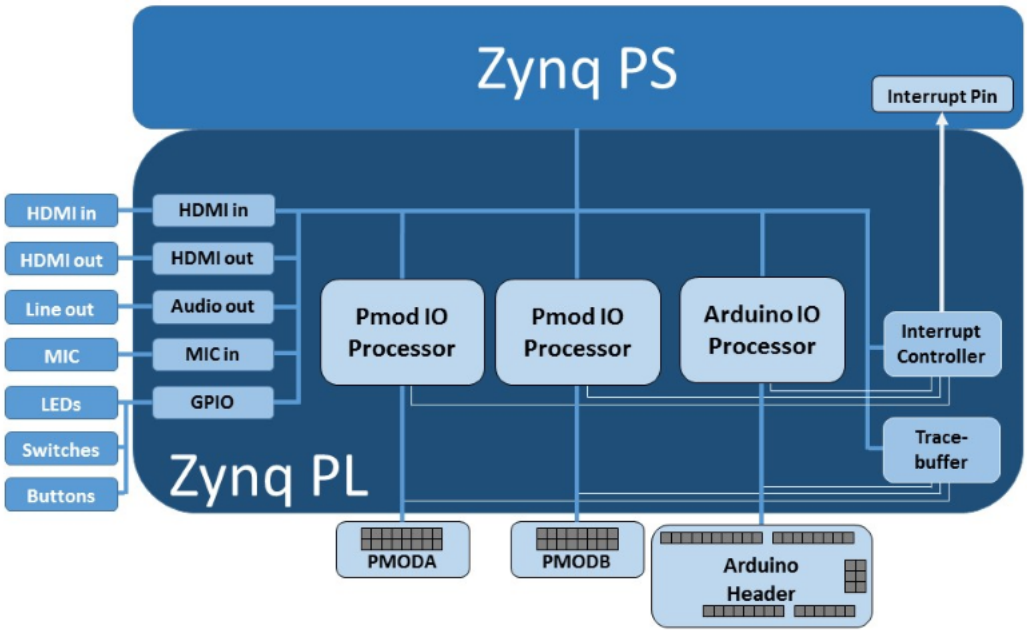
just like a software library. In this example, separate image processing functions could be implemented in different overlays and loaded from Python on demand.

Base Overlay

The base overlay is the default overlay included with the PYNQ-Z1 image, and is automatically loaded into the Programmable Logic when the system boots.

This overlay includes the following hardware:

- HDMI In
- HDMI Out
- Mic in
- Audio out
- User LEDs, Switches, Pushbuttons
- 2x PMOD IOP
- Arduino IOP
- Trace buffer



HDMI

The HDMI controllers are connected directly to the HDMI interfaces. There is no external HDMI circuitry.

Digilent HDMI reference for the PYNQ-Z1 board

Both HDMI interfaces are connected to DDR memory. Video can be streamed from the HDMI *in* to memory, and from memory to HDMI *out*. This allows processing of video data from python, or writing an image or Video stream from Python to the HDMI out.

Note that Jupyter notebooks supports embedded video. However, video captured from the HDMI will be in raw format and would not be suitable for playback in a notebook without appropriate encoding.

HDMI In

The HDMI in interface can capture standard HDMI resolutions. After a HDMI source has been connected, and the HDMI controller started, it will automatically detect the incoming data. The resolution can be read from the HDMI Python class, and the image data can be streamed to DDR memory.

HDMI Out

The HDMI out IP supports the following resolutions:

- 640x480
- 800x600
- 1024x768
- 1280x1024
- 1920x1080

Data can be stream from DDR memory to the HDMI output. The Pynq HDMI Out python instance contains frame-buffers to allow for smooth display of video data.

See the 5_base_overlay_video.ipynb notebook in the getting started directory for examples of using the HDMI In and Out.

Mic in

The PYNQ-Z1 board has an integrated mic on the board and is connected directly to the Zynq PL pins. This means the board does not have an audio codec. The Mic generates audio data in PDM format.

Digilent MIC in reference for the PYNQ-Z1 board

Audio out

The audio out IP is connected to a standard 3.5mm audio jack on the board. The audio output is PWM driven mono.

Digilent Audio Out reference for the PYNQ-Z1 board

User IO

The PYNQ-Z1 board includes two tri-color LEDs, 2 switches, 4 push buttons, and 4 individual LEDs. In the base overlay, the user IO is connected directly to the PS, and can be controlled directly from Python.

IOPs

IOPs are dedicated IO processor subsystems that allow peripherals with different IO standards to be connected to the system on demand. This allows a software programmer to use a wide range of peripherals with different interfaces and protocols. The same overlay can be used to support different peripheral.

There are two types of IOPs: Pmod, and Arduino. Both types of IOPs have a similar architecture, but have different configurations of IP to connect to supported peripherals.

Pmods are covered in more detail in the next section.

Trace buffer

A trace buffer is available and can be used to capture trace data on the Pmod, and Arduino interfaces for debug. The trace buffer is connected directly to DDR. This allows trace data on the interfaces to be streamed back to DDR memory for analysis in Python.

Trace data can be displayed as decoded waveforms inside a Jupyter notebook.

Examples

See the trace buffer_i2c.ipynb and the trace buffer_spi.ipynb in the examples directory for examples on how to use the trace buffer.

CHAPTER 8

Peripherals and Interfaces

Table of Contents

Introduction

The Zynq PL can support many types of protocols and interfaces for external peripherals. The Pynq-Z1 has two Pmod ports and one Arduino interface for connecting external peripherals directly to the Zynq PL. This allows peripherals to be controlled in hardware. Other peripherals can be connected to these ports via adapters, or with a breadboard.

The USB port can also be used to connect standard USB peripherals to the Zynq PS. Linux drivers are required for connecting USB peripherals. The PYNQ image currently includes drivers for some webcams, and USB wifi peripherals.

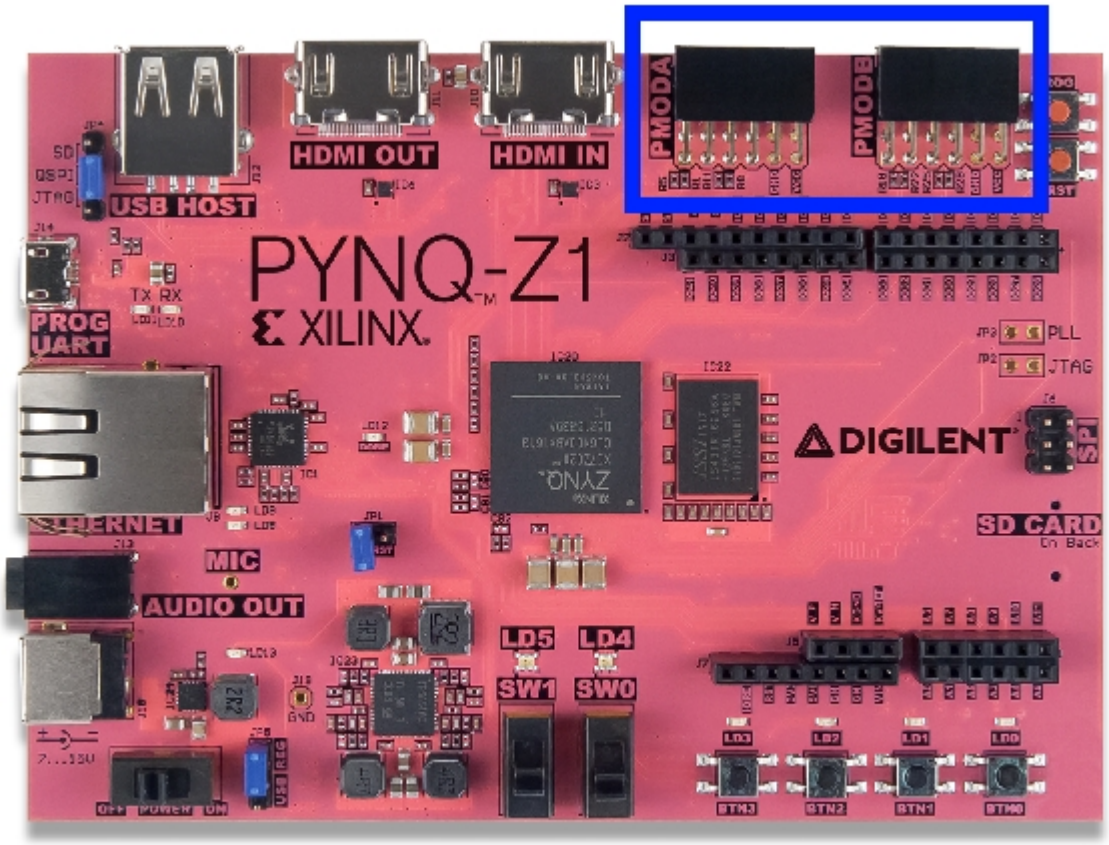
Note that the Zynq PS has dedicated peripherals including Ethernet, USB, UART, IIC, SPI, CAN controllers and GPIO. Only the Ethernet, USB, and UART are connected externally on the board. It is possible to connect the other controllers internally to the Zynq PL. The peripherals could then be used internally inside the PL, or routed to PL pins. E.g. to Pmod, or Arduino pins. This would require an overlay design.

In the base overlay, each IOP has its only set of controllers, implemented in programmable logic, and does not need to use the Zynq PS peripherals.

Pmod port

A Pmod port is an open 12-pin interface that is supported by a range of Pmod peripherals from Digilent and third party manufacturers. Typical Pmod peripherals include sensors (voltage, light, temperature), communication interfaces (Ethernet, serial, wifi, bluetooth), and input and output interfaces (buttons, switches, LEDs).

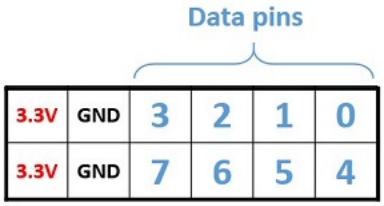
There are two Pmod connectors on PYNQ-Z1.



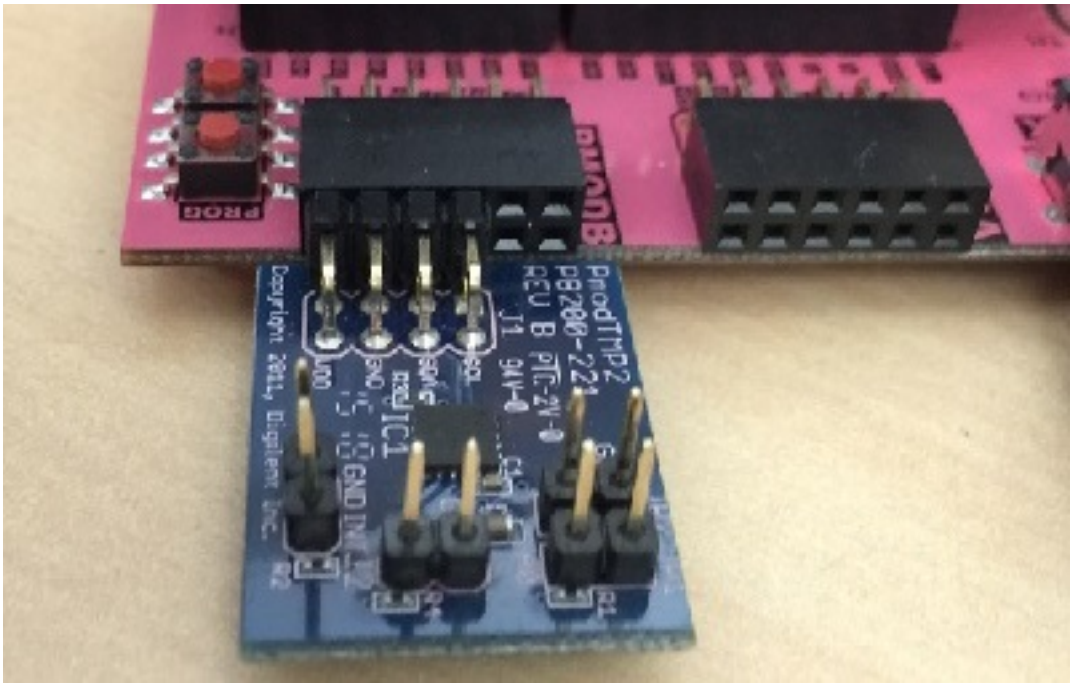
Pmod pins

Each Pmod connector has 12 pins arranged in 2 rows of 6 pins. Each row has 3.3V (VCC), ground (GND) and 4 data pins. Using both rows gives 8 data pins in total.

Pmods come in different configurations depending on the number of data pins required. e.g. Full single row: 1x6 pins; full double row: 2x6 pins; and partially populated: 2x4 pins.



Pmods that use both rows (e.g. 2x4 pins, 2x6 pins), should usually be aligned to the left of the connector (to align with VCC and GND). VCC and GND are labelled on the PYNQ-Z1 board.



Pmod peripherals with only a single row of pins can be connected to either the top row or the bottom row of a Pmod port (again, aligned to VCC/GND). If you are using an existing driver/overlay, you will need to check which pins/rows are supported for a given overlay, as not all options may be implemented. e.g. the Pmod ALS is currently only supported on the top row of a Pmod port, not the bottom row.

Pmod IO standard

All pins operate at 3.3V. Due to different pull-up/pull-down I/O requirements for different peripherals (e.g. IIC requires pull-up, and SPI requires pull-down) the Pmod data pins have different IO standards.

0,1 and 4,5 are connected to pins with pull-down resistors. This can support the SPI interface, and most peripherals. 2,3 and 6,7 are connected to pins with pull-up resistors. This can support the IIC interface.

Pmods already take this pull up/down convention into account in their pin layout, so no special attention is required when using Pmods.

Other Peripherals

Pmod ports are design for use with Pmods. The 8 data pins of a Pmod port can be used to connect to a breadboard, or directly to other peripherals.

Grove peripherals which use a four wire interface can also be connected to the Pmod port through a *PYNQ Grove Adapter*.

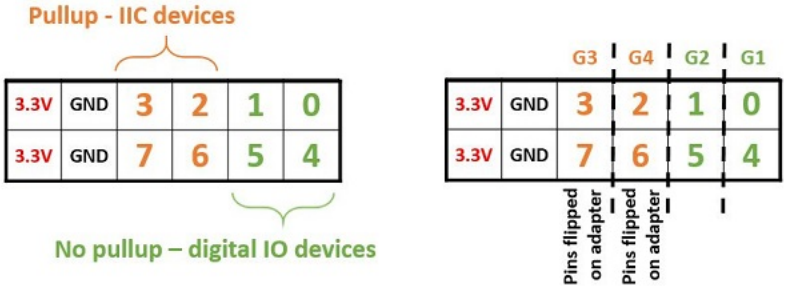
PYNQ Grove Adapter

A Grove connector has four pins, VCC and GND, and two data pins.

The PYNQ Grove Adapter has four connectors (G1 - G4), allowing up to four Grove devices to be connected to one Pmod port. Remember that an IOP application will be required to support the configuration of connected peripherals.

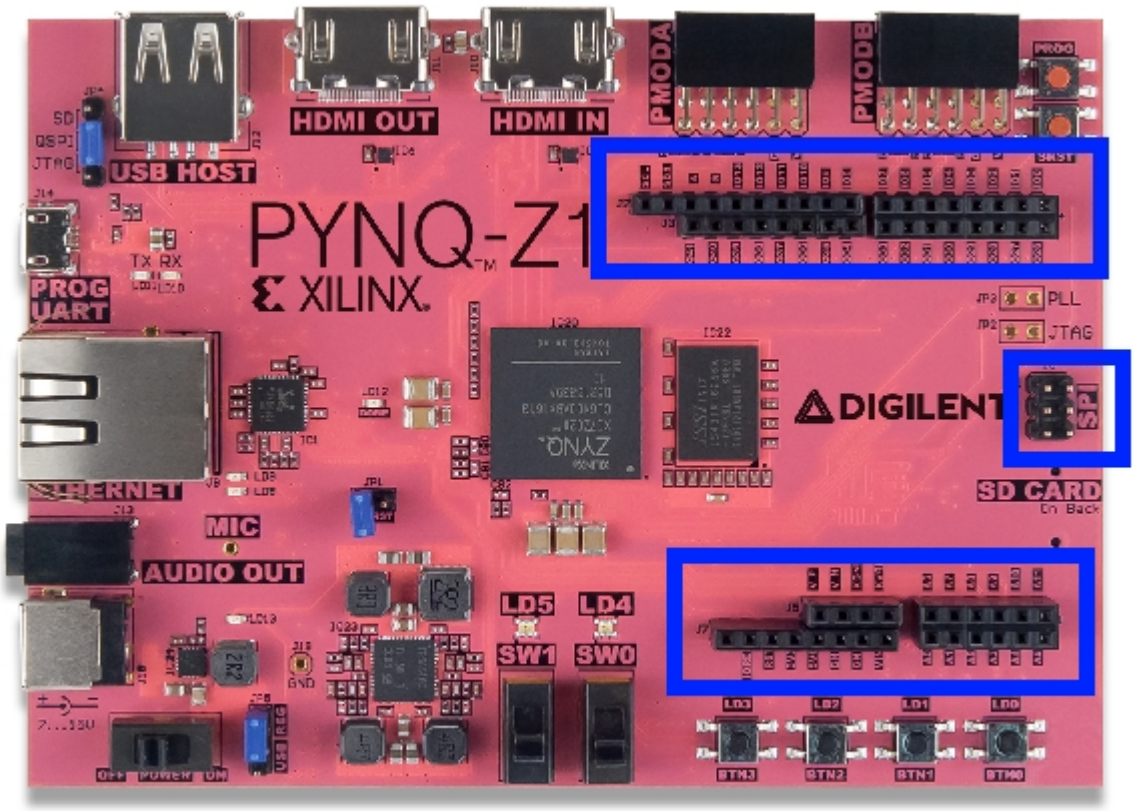
Pmod IO standard for Grove

On the grove adapter G1 and G2 map to Pmod pins [0,4] and [1,5], which are connected to pins with pull-down resistors (supports SPI interface, and most peripherals). G3 and G4 map to pins [2,6], [3,7], which are connected to pins with pull-up resistors (IIC), as indicated in the image.



Arduino connector

There is one Arduino connector on the board and can be used to connect to Arduino compatible shields.



Arduino pins

Each Arduino connector has 6 analog pins (A0 - A5), 14 multi-purpose Digital pins (D0 - D13), 2 dedicated I2C pins (SCL, SDA), and 4 dedicated SPI pins.

ChipKit pins

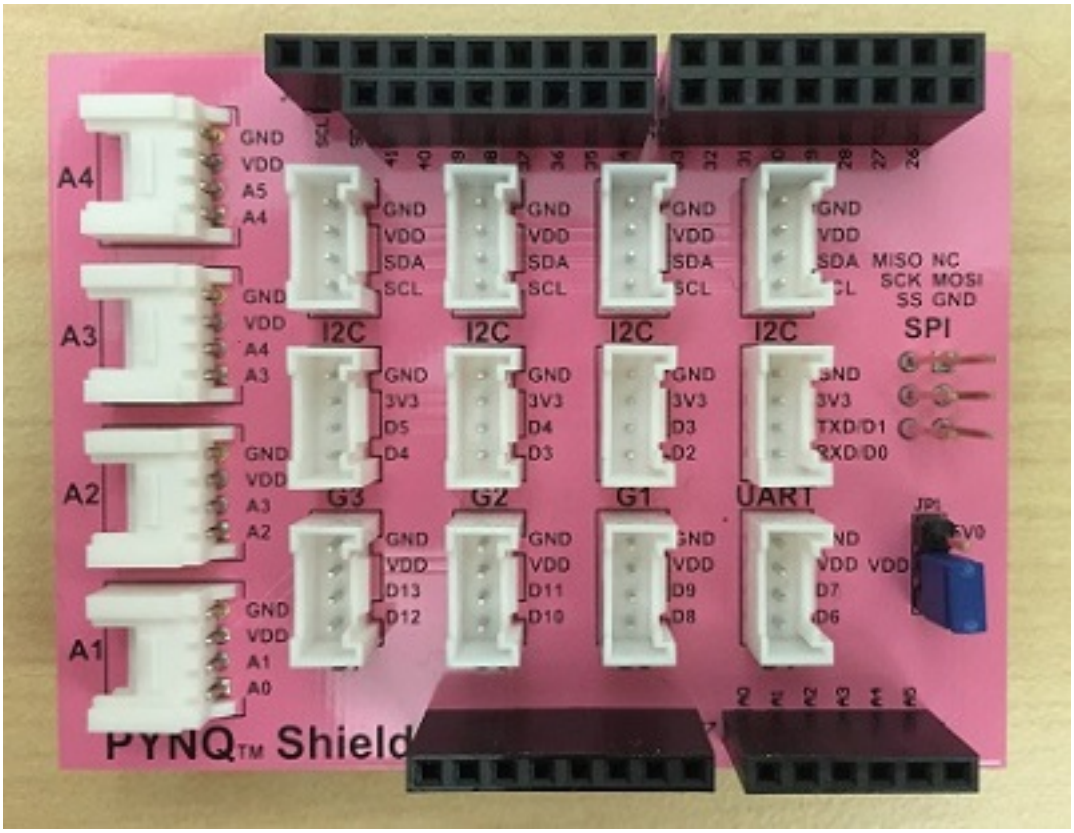
There are also *ChipKit* pins beside the Arduino interface on the PYNQ-Z1 board. These pins are connected to the Zynq PL, but are not enabled in the base overlay.

Supported Peripherals

Most Arduino compatible shields can be used with the PYNQ-Z1 board. However, the PYNQ-Z1 board has a limited analog range, so not all Arduino analog shields are supported.

PYNQ Shield

Each Grove connector has 4 pins. The PYNQ Shield connects to the Arduino and ChipKit pins on the PYNQ-Z1 board. The PYNQ shield has 12 Grove connectors for digital IO (I2C, UART, G1 - G7) and 4 Grove connectors for analog IO (A1 - A4).



With the PYNQ shield jumper (JP1) set to 3.3V (as in the figure), all the pins operate at 3.3V. With JP1 set to 5V, G4 - G7 operated at VDD = 5V.

The Arduino pins, and ChipKit pins are also passed to the top of the board to allow additional shields to be attached.

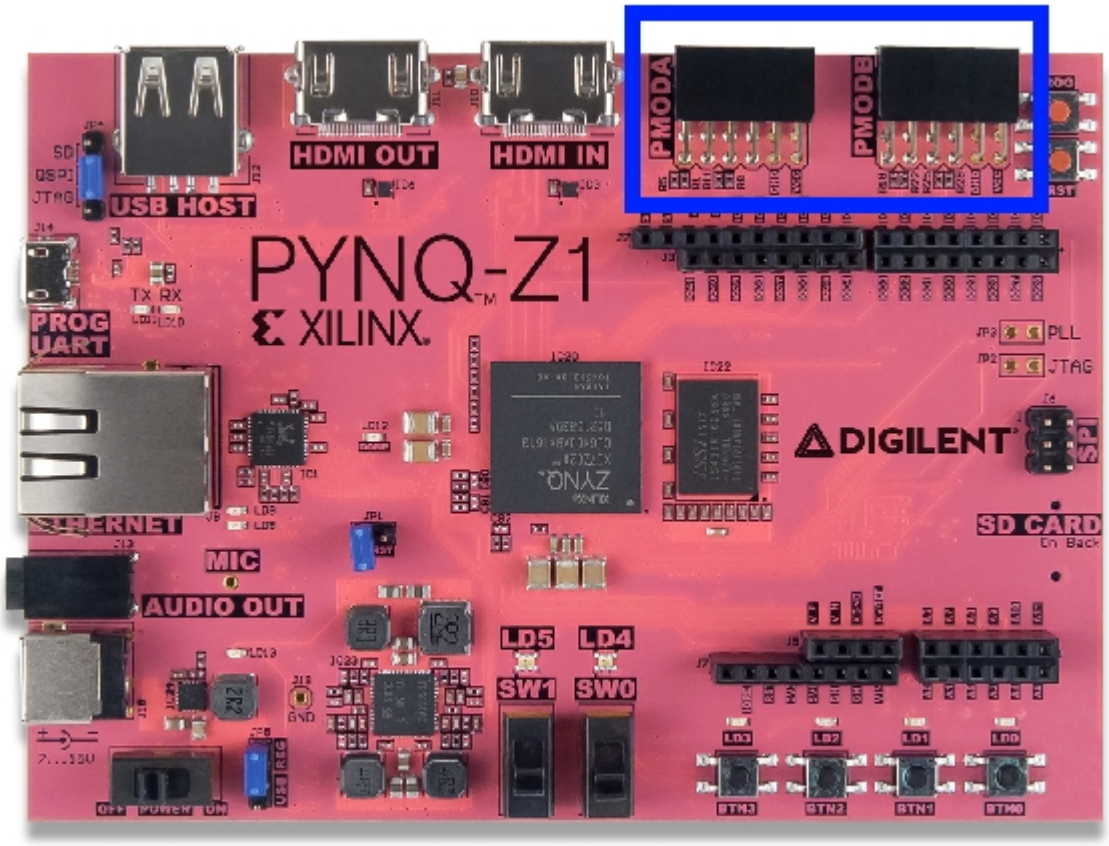
Using Peripherals

Pynq introduces IOPs (Input Output Processors) which are covered in the next section. An IOP consists of a MicroBlaze processor with dedicated peripherals which can be selected and routed to the physical interface at runtime. An IOP provides flexibility allowing peripherals with different protocols and interfaces to be used with the same overlay. A peripheral will have an IOP driver application, and a Python wrapper. The next sections will cover the IOP architecture, and how to write driver applications and the corresponding Python wrapper for a peripheral.

Using Peripherals with the Base overlay

Base overlay

The PYNQ-Z1 has 2 Pmod connectors. PMODA and PMODB as indicated below are connected to the FPGA fabric.



Using Pmods with an overlay

To use a peripheral two software components are required; a driver application written in C for the IOP, and a Python module. These components are provided as part of the Pynq package for supported peripherals. See the *IO Processors: Writing your own software* section of the documentation for writing drivers for your own peripherals.

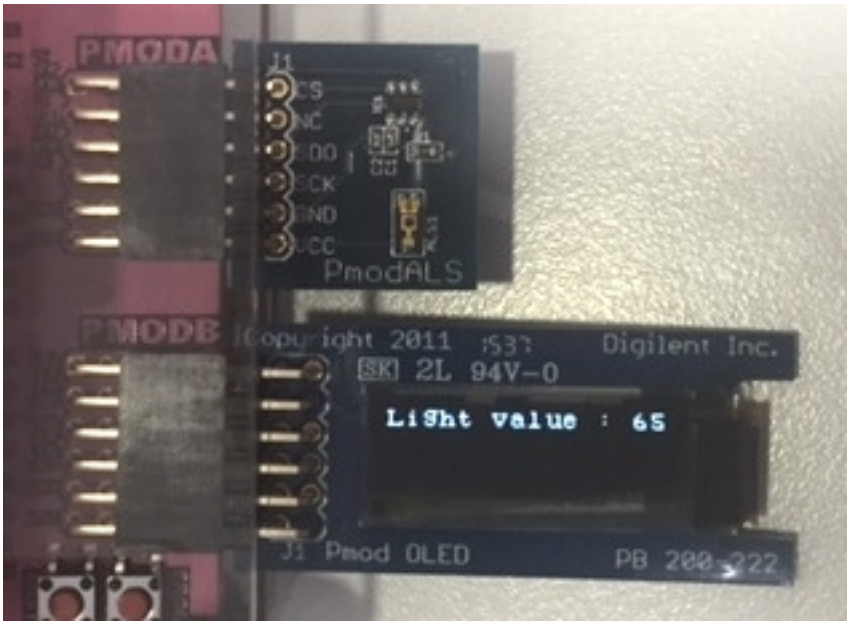
The Python module instantiates the peripheral, and loads the driver application to the appropriate IOP. The IOP will also be reset and start executing the new application.

The Python module will send commands which the IOP will interpret and execute. The Python module may also send the data if necessary. The IOP will read from and write data into the shared memory area.

Example: Using the OLED and the Ambient Light Sensor (ALS)

This examples requires the PmodOLED (OLED), and PmodALS (Ambient Light Sensor). Plug the PmodALS into PMODA, and PmodOLED into the top row of PMODB. (Currently, the PmodALS can only be used in the top row of a Pmod port.)

OLED displaying light reading from ambient light sensor:



Execute the next cell to load the FPGA fabric with the desired overlay, and then import the OLED module and instantiate it on PMODB:

```
In [1]: from pynq import Overlay
        from pynq.iop import Pmod_OLED
        from pynq.iop import PMODB

        ol = Overlay("base.bit")
        ol.download()
        oled = Pmod_OLED(PMODB)
```

Try writing a message to the OLED.

```
In [2]: oled.write("Hello World")
```

```
In [3]: oled.clear()
```

Import the ALS library, create an instance of the ALS Pmod, and read the value from the sensor.

```
In [4]: from pynq.iop import Pmod_ALS
        from pynq.iop import PMODA

        als = Pmod_ALS(PMODA)
        als.read()
```

Write the value from the ALS to the OLED. The ALS sensor returns an 8-bit value.

- 0 : Darkest
- 255 : Brightest

```
In [5]: oled.write("Light value : " + str(als.read()))
```

```
In [6]: import time
        from pynq.iop import Pmod_ALS
        from pynq.iop import PMODA

        als = Pmod_ALS(PMODA)

        als.set_log_interval_ms(100)
        als.start_log()
        time.sleep(1)
        als.stop_log()
        als.get_log()
```

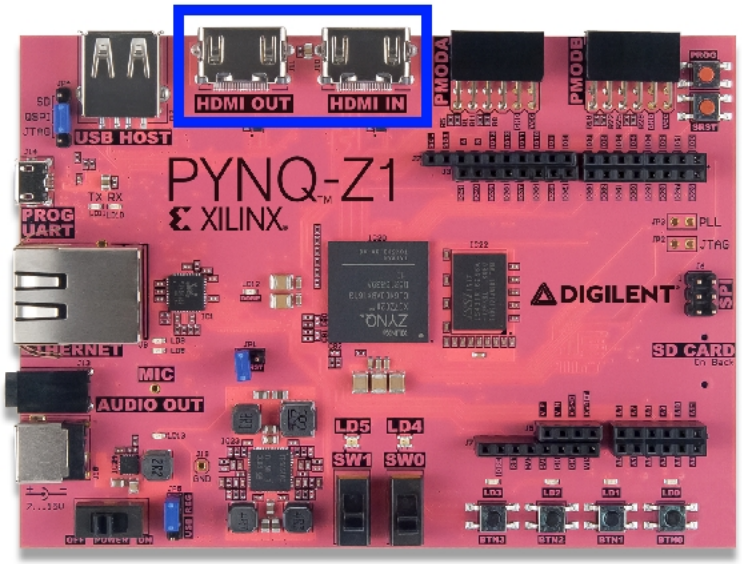
For information on other supported peripherals and their API, see the *pynq.iop package* section of the documentation.

Video using the Base Overlay

The PYNQ-Z1 board contains a HDMI input port, and a HDMI output port connected to the FPGA fabric of the Zynq® chip. This means to use the HDMI ports, HDMI controllers must be included in a hardware library or overlay.

The base overlay contains a HDMI input controller, and a HDMI Output controller, both connected to their corresponding HDMI ports. A frame can be captured from the HDMI input, and streamed into DDR memory. The frames in DDR memory, can be accessed from Python.

A framebuffer can be shared between HDMI in and HDMI out to enable streaming.



Video IO

The overlay contains two video controllers, HDMI in and out. Both interfaces can be controlled independently, or used in combination to capture an image from the HDMI, process it, and display it on the HDMI out.

There is also a USB controller connected to the Zynq PS. A webcam can also be used to capture images, or video input, that can be processed and displayed on the HDMI out.

The HDMI video capture controller

To use the HDMI in controller, connect the on-board HDMI In port to a valid video source. E.g. your laptop can be used if it has HDMI out. Any HDMI video source can be used up to 1080p.

To use the HDMI in, ensure you have connected a valid HDMI source and execute the next cell. If a valid HDMI source is not detected, the HDMI in controller will timeout with an error.

```
In [1]: from pynq import Overlay
        from pynq.drivers.video import HDMI

        # Download bitstream
        Overlay("base.bit").download()

        # Initialize HDMI as an input device
        hdmi_in = HDMI('in')
```

The HDMI() argument 'in' indicates that the object is in capture mode.

When a valid video input source is connected, the controller should recognize it and start automatically. If a HDMI source is not connected, the code will time-out with an error.

Starting and stopping the controller

You can manually start/stop the controller

```
In [2]: hdmi_in.start()
```

```
In [3]: hdmi_in.stop()
```

Readback from the controller

To check the state of the controller:

```
In [4]: state = hdmi_in.state()
        print(state)
```

```
2
```

The state is returned as an integer value, with one of three possible values:

- 0 if disconnected
- 1 if streaming
- 2 if paused

You can also check the width and height of the input source (assuming a source is connected):

```
In [5]: hdmi_in.start()

        width = hdmi_in.frame_width()
        height = hdmi_in.frame_height()
        print('HDMI is capturing a video source of resolution {}x{}'\
              .format(width,height))
```

```
HDMI is capturing a video source of resolution 1920x1080
```

HDMI Frame list

The HDMI object holds a frame list, that can contain up to 3 frames, and is where the controller stores the captured frames. At the object instantiation, the current frame is the one at index 0. You can check at any time which frame index is active:

```
In [6]: hdmi_in.frame_index()
```

```
Out[6]: 0
```

The `frame_index()` method can also be used to set a new index, if you specify an argument with the method call. For instance:

```
In [7]: index = hdmi_in.frame_index()
        hdmi_in.frame_index(index + 1)
```

This will set the current frame index to the next in the sequence. Note that, if `index` is 2 (the last frame in the list), `(index+1)` will cause an exception.

If you want to set the next frame in the sequence, use:

```
In [8]: hdmi_in.frame_index_next()
```

```
Out[8]: 2
```

This will loop through the frame list and it will also return the new index as an integer.

Access the current frame

There are two ways to access pixel data: `hdmi.frame()` and `hdmi.frame_raw()`.

```
In [9]: from IPython.display import Image

        frame = hdmi_in.frame()
        orig_img_path = '/home/xilinx/jupyter_notebooks/Getting_Started/images/hdmi_in_fra
        frame.save_as_jpeg(orig_img_path)
        Image(filename=orig_img_path)
```

This will dump the frame as a list `_frame[height, width][rgb]`. Where `rgb` is a tuple `(r,g,b)`. If you want to modify the green component of a pixel, you can do it as shown below. In the example, the top left quarter of the image will have the green component increased.

```
In [10]: for x in range(int(width/2)):
             for y in range(int(height/2)):
                 (red,green,blue) = frame[x,y]
                 green = green*2
                 if(green>255):
                     green = 255
                 frame[x,y] = (red, green, blue)

         new_img_path = '/home/xilinx/jupyter_notebooks/Getting_Started/images/hdmi_in_fra
         frame.save_as_jpeg(new_img_path)
         Image(filename=new_img_path)
```

This frame() method is a simple way to capture pixel data, but processing it in Python will be slow. If you want to dump a frame at a specific index, just pass the index as an argument of the `frame()` method:

```
In [11]: # dumping frame at index 2
         frame = hdmi_in.frame(2)
```

If higher performance is required, the `frame_raw()` method can be used:

```
In [12]: # dumping frame at current index
         frame_raw = hdmi_in.frame_raw()

         # dumping frame at index 2
         frame_raw = hdmi_in.frame_raw(2)
```

This method will return a fast memory dump of the internal frame list, as a mono-dimensional list of dimension `frame[1920*1080*3]` (This array is of fixed size regardless of the input source resolution). 1920x1080 is the maximum supported frame dimension and 3 separate values for each pixel (Blue, Green, Red).

When the resolution is less than 1920x1080, the user must manually extract the correct pixel data.

For example, if the resolution of the video input source is 800x600, meaningful values will only be in the range `frame_raw[1920*i*3]` to `frame_raw[(1920*i + 799)*3]` for each `i` (rows) from 0 to 599. Any other position outside of this range will contain invalid data.

```
In [13]: # printing the green component of pixel (0,0)
         print(frame_raw[1])

         # printing the blue component of pixel (1,399)
         print(frame_raw[1920 + 399 + 0])

         # printing the red component of the last pixel (599,799)
         print(frame_raw[1920*599 + 799 + 2])
```

```
175
227
```

```
249
```

Frame Lists

To draw or display smooth animations/video, note the following:

Draw a new frame to a frame location not currently in use (an index different to the current `hdmi.frame_index()`). Once finished writing the new frame, change the current frame index to the new frame index.

The HDMI out controller

Using the HDMI output is similar to using the HDMI input. Connect the HDMI OUT port to a monitor, or other display device.

To instantiate the HDMI controller:

```
In [14]: from pynq.drivers import HDMI

         hdmi_out = HDMI('out')
```

For the HDMI controller, you have to start/stop the device explicitly:

```
In [15]: hdmi_out.start()
```

```
In [16]: hdmi_out.stop()
```

To check the state of the controller:

```
In [17]: state = hdmi_out.state()
         print(state)
```

```
0
```

The state is returned as an integer value, with 2 possible values:

- 0 if stopped
- 1 if running

After initialization, the display resolution is set at the lowest level: 640x480 at 60Hz.

To check the current resolution:

```
In [18]: print(hdmi_out.mode())
```

```
640x480@60Hz
```

This will print the current mode as a string. To change the mode, insert a valid index as an argument when calling mode():

```
In [19]: hdmi_out.mode(4)
```

```
Out[19]: '1920x1080@60Hz'
```

Valid resolutions are:

- 0 : 640x480, 60Hz
- 1 : 800x600, 60Hz
- 2 : 1280x720, 60Hz
- 3 : 1280x1024, 60Hz

- 4 : 1920x1080, 60Hz

Input/Output Frame Lists

To draw or display smooth animations/video, note the following:

Draw a new frame to a frame location not currently in use (an index different to the current `hdmi.frame_index()`). Once finished writing the new frame, change the current frame index to the new frame index.

Streaming from HDMI Input to Output

To use the HDMI input and output to capture and display an image, make both the HDMI input and output share the same frame list. The frame list in both cases can be accessed. You can make the two object share the same frame list by a frame list as an argument to the second object's constructor.

```
In [20]: from pynq.drivers.video import HDMI

         hdmi_in = HDMI('in')
         hdmi_out = HDMI('out', frame_list=hdmi_in.frame_list)
         hdmi_out.mode(4)
```

```
Out[20]: '1920x1080@60Hz'
```

To start the controllers:

```
In [21]: hdmi_out.start()
         hdmi_in.start()
```

The last step is always to stop the controllers and delete HDMI objects.

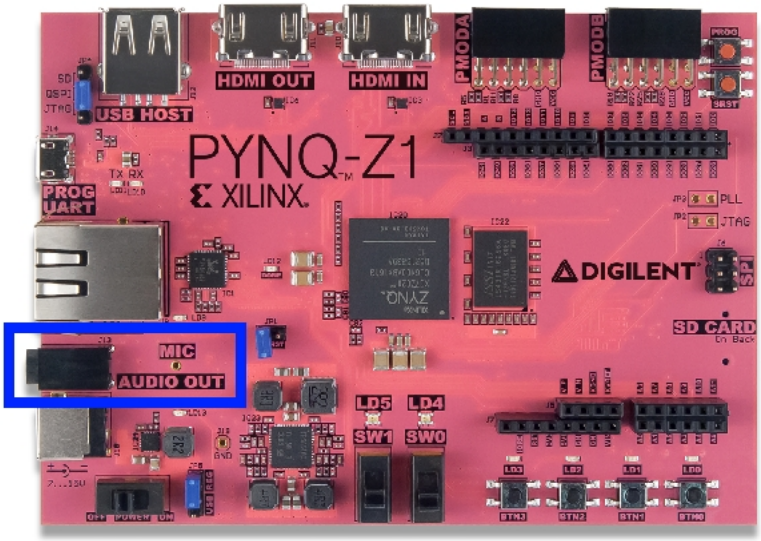
```
In [22]: hdmi_out.stop()
         hdmi_in.stop()
         del hdmi_out
         del hdmi_in
```

Audio using the Base Overlay

The PYNQ-Z1 board contains an integrated MIC, and line out connected to a 3.5mm jack. Both these interfaces are connected to the FPGA fabric of the Zynq® chip. The Microphone has a PDM interface, and the line out is a PWM driven mono output.

It is possible to play back audio from the board in a notebook, and to capture audio from other interfaces like HDMI, or a USB audio capture device. This notebook will only consider the MIC and line out interfaces on the board.

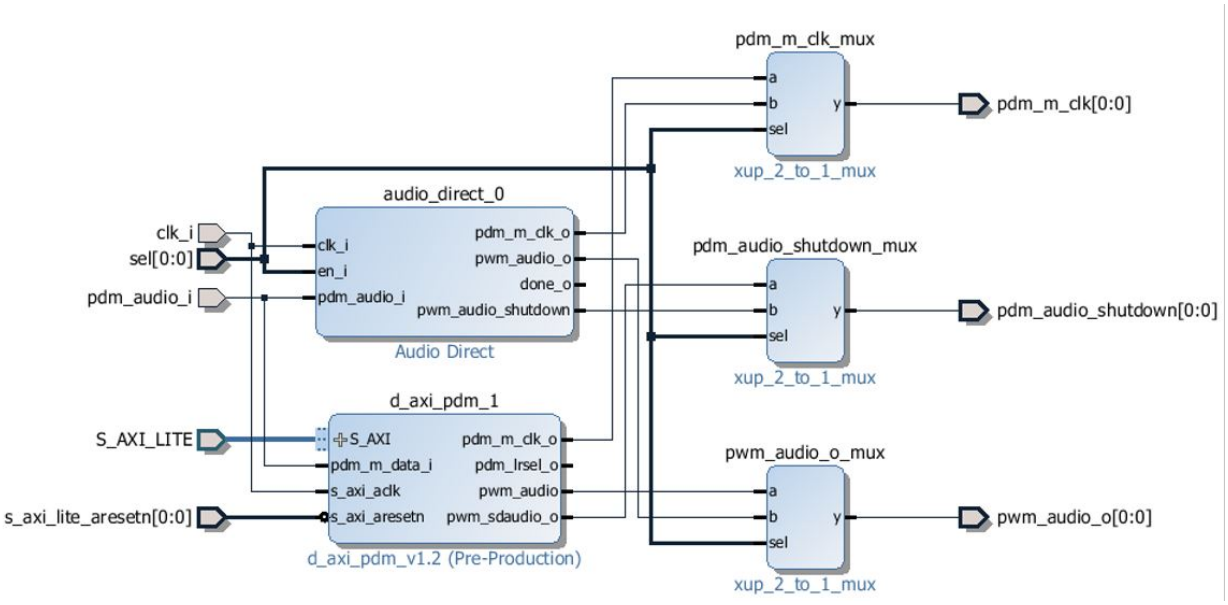
The Microphone is integrated onto the board, as indicated in the image below. The MIC hole should not be covered when capturing audio.



Audio IP in base overlay

To use audio on the PYNQ-Z1, audio controllers must be included in a hardware library or overlay. The *base* overlay contains a the PDM capture and PWM driver for the two audio interfaces as indicated in the image below:

The Audio IP in the *base* overlay consists of a PDM block to interface the MIC, and an *Audio Direct* IP block to drive the line out (PWM). There are three multiplexors. This allows the line out to be driven from the PS, or the MIC can be streamed directly to the output. The line out can also be disabled.



Using the MIC

To use the MIC, first create an instance of the Audio class. The audio class can be used to access both the MIC and the line out.

```
In [1]: from pynq.drivers import Audio
        audio = Audio()
```

Capture audio

Capture a 4 second sample from the microphone, and save the raw pdm file to disk:

```
In [2]: # Record a sample
        audio.record(4)
        # Save recorded sample
        audio.save("Recording_1.pdm")
```

Playback on the board

Connect headphones, or speakers to the 3.5mm line out and playback the captured audio:

```
In [3]: # Play recorded sample
        audio.play()
```

You can also playback from a pre-recorded pdm file

```
In [4]: # Load a sample
        audio.load("/home/xilinx/pynq/drivers/tests/pynq_welcome.pdm")
        # Play loaded sample
        audio.play()
```

CHAPTER 12

IO Processor Architecture

Table of Contents

Introduction

For overlays to be useful, they must provide sufficient functionality, while also providing flexibility to suit a wide range of applications. Flexibility in the base overlay is demonstrated through the use of IO Processors (IOPs).

An IO Processor is implemented in the programmable logic and connects to and controls an external port on the board. There are two types of IOP: Pmod IOP and Arduino IOP.

Each IOP contains a MicroBlaze processor, a configurable switch, peripherals, and local memory for the MicroBlaze instruction and data memory. The local memory is dual-ported (implemented in Xilinx BRAMs), with one port connected to the MicroBlaze, and the other connected to the ARM® Cortex®-A9 processor. This allows the ARM processor to access the MicroBlaze memory and dynamically write a new program to the MicroBlaze instruction area.

The data area of the memory can be used for communication and data exchanges between the ARM processor and the IOP(s). E.g. a simple mailbox.

The IOP also has an interface to DDR memory. This allows the DDR to be used as data memory in addition to the local memory. This allows larger applications to be written (where data memory is the limitation) and allows a larger mailbox size for data transfer between the PS (Python) and the IOP. The DDR interface also allows different IOPs to communicate with each other directly without intervention from the PS (Python).

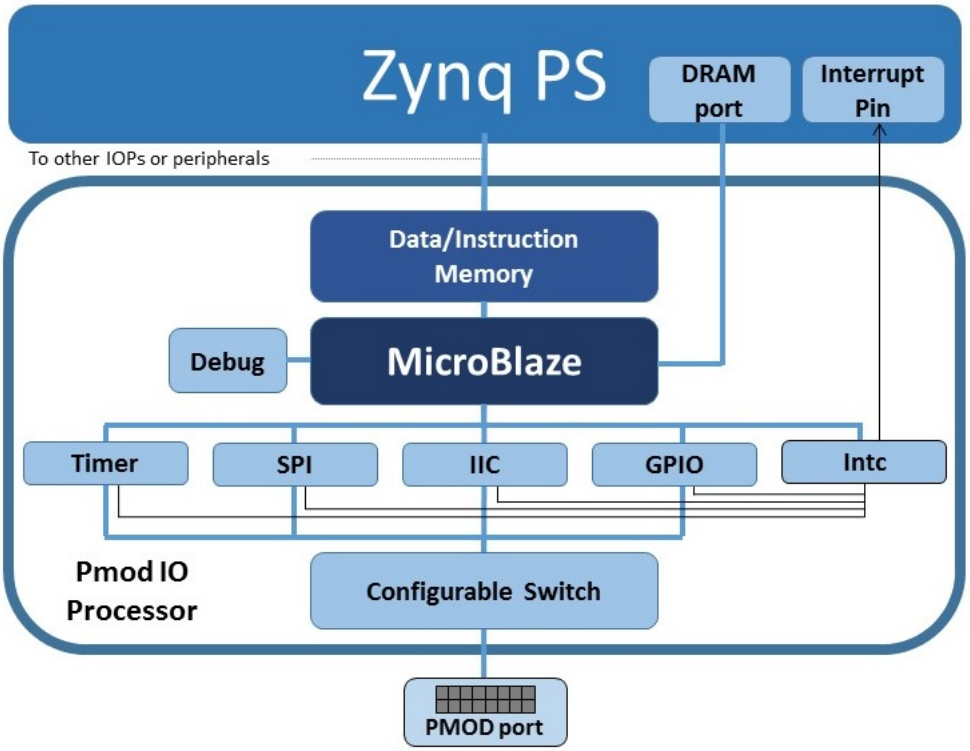
In the base overlay, two IOPs control each of the two Pmod interfaces, and another IOP controls the Arduino interface. Inside the IOP are dedicated peripherals; timers, UART, IIC, SPI, GPIO, and a configurable switch. (Not all peripherals are available in the Pmod IOP.)

IIC and SPI are standard interfaces used by many of the available Pmod, Grove and other peripherals. GPIO can be used to connect to custom interfaces or used as simple inputs and outputs.

When a Pmod, Arduino shield, or other peripheral is plugged in to a port, the configurable switch allows the signals to be routed dynamically to the required dedicated interface. This is how the IOP provides flexibility and allows peripherals with different pin connections and protocols to be used on the same port.

Pmod IOP

Two Pmod IOPs are included in the base overlay to control each of the two Pmod interfaces on the board.



As indicated in the diagram, the Pmod IOP has a MicroBlaze, a configurable switch, and the following peripherals:

- AXI Timer
- AXI IIC
- AXI SPI
- AXI GPIO

Pmod IOP peripherals

I2C

The I2C configuration is:

- Frequency: 100KHz
- Address mode: 7 bit

SPI

The SPI configuration is:

- Standard mode
- Transaction width: 8

- Frequency: 6.25 MHz (100MHz/16)
- Master mode
- Fifo depth: 16

GPIO blocks

The GPIO block supports 8 input or output pins

Timer

The timer is 32 bits width, and has a *Generate* output, and a PWM output. The *Generate* signal can output one-time or periodic signal based on a loaded value. For example, on loading a value, the timer can count up or down. Once the counter expires (on a carry) a signal can be generated. The timer can stop, or automatically reload.

Interrupt controller

The I2c, SPI, GPIO and Timer are connected to the interrupt controller. This is the standard MicroBlaze interrupt controller, and interrupts can be managed by the IOP in a similar way to any other MicroBlaze application.

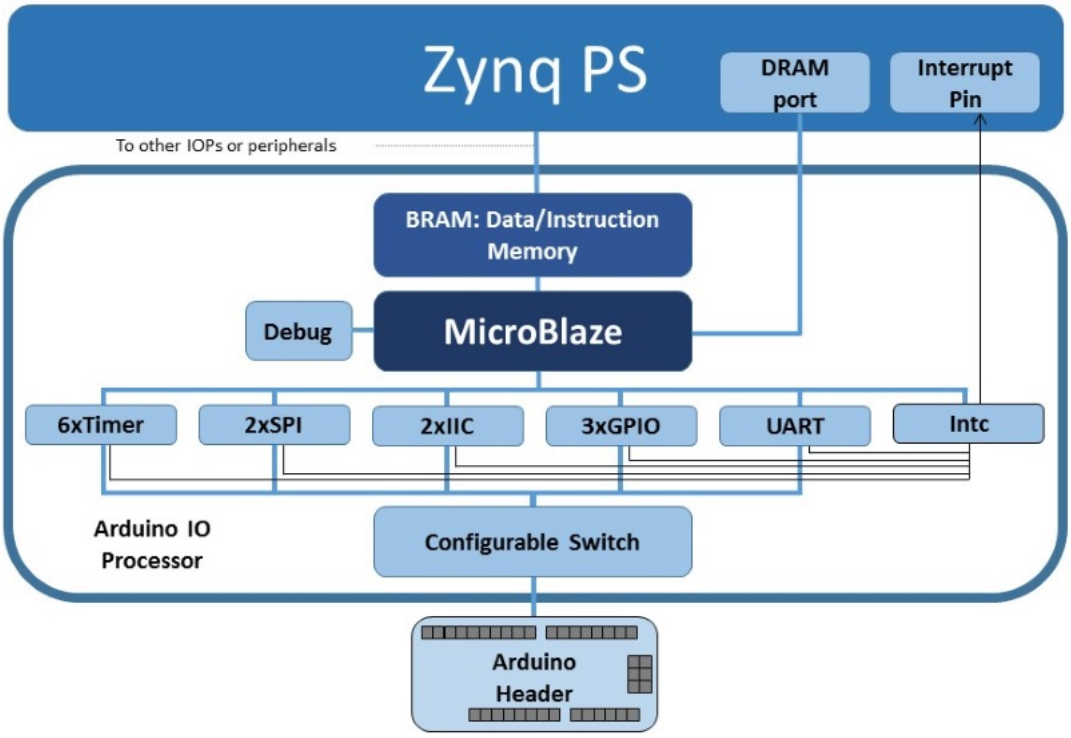
Pmod IOP configurable switch

The configurable switch can route signals from the internal IP to the external FPGA pins. The switch is controlled by the MicroBlaze, and can be configured from an IOP application.

For details on using the switch, see the next sections on *IO Processors: Writing your own software* and *IO Processors: Using peripherals in your applications*.

Arduino IOP

Similar to the Pmod IOP, an Arduino IOP is available to control the Arduino interface. The Arduino IOP is similar to the PMOD IOP, but has some additional internal peripherals (extra timers, an extra I2c, and SPI, a UART, and an XADC). The configurable switch is also different to the Pmod switch.



As indicated in the diagram, the Arduino IOP has a MicroBlaze, a configurable switch, and the following peripherals:

- 6x AXI Timer
- 2x AXI IIC
- 2x AXI SPI
- 3x AXI GPIO
- 1x AXI UART
- 1x AXI Interrupt controller
- 1x AXI XADC

Arduino IOP peripherals

The I2C, SPI, GPIO and Timer blocks are the same as the Pmod IOP blocks. The only difference in the Arduino IOP with these blocks is that for the IIC and SPI, 2 interfaces are enabled, for the GPIO 3 blocks are include, and 6 timers.

I2C

Two I2C available.

SPI

Two SPI available. One is always connected to the Arduino interface dedicated SPI pins.

GPIO blocks

There are three GPIO block available. They support 16 input or output pins on the Arduino interface (D0 - D15).

Timers

There are six timers available.

UART

There is a UART controller, with a fixed configuration of 9600 baud. The UART can be connected to the Arduino UART pins. The UART configuration is hard coded, and is part of the overlay. It is not possible to modify the UART configuration in software.

Interrupt controller

The interrupt controller can be connected to all the analog and digital pins, and each of the 6 timers, the I2Cs, the SPIs, the XADC, and UART. This means an external pin on the shield interface can trigger an interrupt. An internal peripheral can also trigger an interrupt.

Arduino shields have fixed possible configurations. According to the Arduino specification, the analog pins can be used as analog, or digital I/O.

Other peripherals can be connected as indicated in the table.

Peripheral	Pins
UART	D0, D1
I2C	A4, A5
SPI*	D10 - D13
PWM	D3, D5, D6, D9, D10, D11
Timer	D3 - D6 and D8 - D11

* There are also dedicated pins for a separate SPI.

For example, a shield with a UART and 5 Digital IO can connect the UART to pins D0, D1, and the Digital IO can be connected to pins D2 - D6.

While there is support for analog inputs via the internal XADC, this only allows inputs of 0-1V. The Arduino interface supports 0-5V analog inputs which is not supported on the PYNQ-Z1.

Arduino IOP configurable Switch

The switch is controlled by the MicroBlaze, and can be configured by writing to its configuration registers from an IOP application.

The dedicated SPI pins that are part of the Arduino interface are always connected to one of the SPI controllers.

The analog and digital pins can be configured by writing a 4-bit value to the corresponding place in the IO switch configuration registers, similar to the Pmod switch.

For details on using the switch, see the next sections on *IO Processors: Writing your own software* and *IO Processors: Using peripherals in your applications*.

CHAPTER 13

IO Processors: Software Architecture

Table of Contents

There are a number of steps required before you can start writing your own software for an IOP (IO Processor). This document will describe the IOP architecture, and how to set up and build the required software projects to allow you to write your own application for the MicroBlaze inside an IOP. Xilinx® SDK projects can be created manually using the SDK GUI, or software can be built using a Makefile flow.

IO Processors

As described in the previous section, an IOP can be used as a flexible controller for different types of external peripherals. The ARM® Cortex®-A9 is an application processor, which runs Pynq and Jupyter notebook on a Linux OS. This scenario is not well suited to real-time applications, which is a common requirement for an embedded systems. In the base overlay there are three IOPs. As well as acting as a flexible controller, an IOP can be used as dedicated real-time controller.

IOPs can also be used standalone to offload some processing from the main processor. However, note that the MicroBlaze processor inside an IOP in the base overlay is running at 100 MHz, compared to the Dual-Core ARM Cortex-A9 running at 650 MHz. The clock speed, and different processor architectures and features should be taken into account when offloading pure application code. e.g. Vector processing on the ARM Cortex-A9 Neon processing unit will be much more efficient than running on the MicroBlaze. The MicroBlaze is most appropriate for low-level, background, or real-time applications.

Software requirements

Xilinx SDK (Software Development Kit) contains the MicroBlaze cross-compiler which can be used to build software for the MicroBlaze inside an IOP. SDK is available for free as part of the Xilinx Vivado WebPack.

The full source code for all supported IOP peripherals is available from the project GitHub. Pynq ships with precompiled IOP executables to support various peripherals (see Pynq Modules), so Xilinx software is only needed if you intend to modify existing code, or build your own IOP applications/peripheral drivers.

The current Pynq release is built using Vivado and SDK 2016.1. it is recommended to use the same version to rebuild existing Vivado and SDK projects. If you only intend to build software, you will only need to install SDK. The full Vivado and SDK installation is only required to modify or design new overlays. Download Xilinx Vivado and SDK 2016.1 You can use the Vivado HLx Web Install Client and select SDK and/or Vivado during the installation.

Compiling projects

Software executables run on the MicroBlaze inside an IOP. Code for the MicroBlaze can be written in C or C++ and compiled using Xilinx SDK .

You can pull or clone the Pynq GitHub repository, and all the driver source and project files can be found in `<GitHub Repository>\Pynq-Z1\sdk`, (Where `<GitHub Repository>` is the location of the PYNQ repository).

SDK Application, Board Support Package, Hardware Platform

Each SDK application project requires a BSP project (Board Support Package), and a hardware platform project. The application project will include the user code (C/C++). The Application project is linked to a BSP. The BSP (Board Support Package) contains software libraries and drivers to support the underlying peripherals in the system and is automatically generated in SDK. The BSP is then linked to a Hardware Platform. A Hardware Platform defines the peripherals in the IOP subsystem, and the memory map of the system. It is used by the BSP to build software libraries to support the underlying hardware. The hardware platform is also automatically generated in SDK.

All *Application* projects and the underlying BSP and hardware platform can be compiled from the command line using makefiles, or imported into the SDK GUI.

You can also use existing projects as a starting point to create your own project.

Hardware Platform (HDF file)

Before an Application project and BSP can be created or compiled in SDK, a *Hardware Platform* project is required.

A Hardware Description File (.hdf), created by Vivado, is used to create the *Hardware Platfrom* project in SDK.

A precompiled .hdf file is provided, so it is not necessary to run Vivado to generate a .hdf file:

```
<GitHub Repository>/Pynq-Z1/sdk/
```

There will be one hardware platform for each overlay.

Board Support Package

Once the hardware platform has been generated, the BSP can be built. A BSP includes software libraries for peripherals in the system. It must be linked to a Hardware Platform, as this is where the peripherals in the system are defined. Once the BSP has been generated, an application project can then be linked to a BSP, and use the software libraries available in the BSP.

A BSP is specific to a processor subsystem. There can be many BSPs associated with an overlay, depending on the types of processors available in the system. In the case of the base overlay, there are two types of processor subsystems: Pmod IOP and Arduino IOP which both require a BSP. The BSPs will share the same hardware platform.

An application for the Pmod IOP will be linked to the Pmod IOP BSP. As the two Pmod IOPs are identical, an application written for one Pmod IOP can run on the other Pmod IOP. An Arduino application will be linked to the Pmod IOP BSP.

Building the projects

A Makefile to automatically create and build the Hardware Platform and the BSP can be found in the same location as the .hdf file.

```
<GitHub Repository>/Pynq-Z1/sdk/makefile
```

Application projects for peripherals that ship with Pynq (e.g. Pmods and Grove peripherals) can also be found in the same location. Each project is contained in a separate folder.

The makefile uses the .hdf file to create the Hardware Platform. The BSP can then be created. The application projects will also be compiled automatically as part of this process.

The makefile requires SDK to be installed, and can be run from Windows, or Linux.

To run `make` from Windows, open SDK, and choose a temporary workspace (make sure this path is external to the downloaded GitHub repository). From the *Xilinx Tools* menu, select *Launch Shell*

In Linux, open a terminal, and source the SDK tools.

From either the Windows Shell, or the Linux terminal, navigate to the sdk folder in your local copy of the GitHub repository:

cd to `<GitHub Repository>/Pynq-Z1/sdk` and run `make`

```
yunq@xsjpsgv106-48% ls -al
total 772
drwxr-xr-x 23 yunq admin   4096 Aug 26 11:08 .
drwxr-xr-x  6 yunq admin   4096 Aug  3 09:43 ..
drwxr-xr-x  4 yunq admin   4096 Aug  3 09:43 arduino_grove_adc
drwxr-xr-x  4 yunq admin   4096 Aug  3 09:43 arduino_grove_buzzer
drwxr-xr-x  4 yunq admin   4096 Aug  3 09:43 arduino_grove_imu
drwxr-xr-x  4 yunq admin   4096 Aug  3 09:43 arduino_grove_ledbar
drwxr-xr-x  4 yunq admin   4096 Aug  3 09:43 arduino_grove_oled
-rw-r--r--  1 yunq admin 680872 Aug 26 11:08 base.hdf
drwxr-xr-x  2 yunq admin   4096 Aug  3 09:43 bin
drwxr-xr-x  2 yunq admin   4096 Aug  3 09:43 bootbin
-rwxr-xr-x  1 yunq admin    549 Aug  3 09:43 build_xsdk.tcl
drwxr-xr-x  4 yunq admin   4096 Aug  3 09:43 mailbox
-rwxr-xr-x  1 yunq admin   1008 Aug  3 09:43 makefile
drwxr-xr-x  4 yunq admin   4096 Aug  3 09:43 pmod_adc
drwxr-xr-x  4 yunq admin   4096 Aug  3 09:43 pmod_als
drwxr-xr-x  4 yunq admin   4096 Aug  3 09:43 pmod_dac
drwxr-xr-x  4 yunq admin   4096 Aug  3 09:43 pmod_dpot
drwxr-xr-x  4 yunq admin   4096 Aug  3 09:43 pmod_grove_adc
drwxr-xr-x  4 yunq admin   4096 Aug  3 09:43 pmod_grove_buzzer
drwxr-xr-x  4 yunq admin   4096 Aug  3 09:43 pmod_grove_imu
drwxr-xr-x  4 yunq admin   4096 Aug  3 09:43 pmod_grove_ledbar
drwxr-xr-x  4 yunq admin   4096 Aug  3 09:43 pmod_grove_oled
drwxr-xr-x  4 yunq admin   4096 Aug  3 09:43 pmod_oled
drwxr-xr-x  4 yunq admin   4096 Aug  3 09:43 pmod_pwm
drwxr-xr-x  4 yunq admin   4096 Aug  3 09:43 pmod_timer
drwxr-xr-x  4 yunq admin   4096 Aug  3 09:43 pmod_tmp2
yunq@xsjpsgv106-49% source /proj/gsd/vivado/Vivado/2016.1/settings64.csh
yunq@xsjpsgv106-50% make
xsdk -batch -source build_xsdk.tcl
```

```
make[1]: Leaving directory `/home/yunq/Pynq-Python/backup/Pynq-Z1/sdk/arduino_gr
ove_ledbar/Debug'
make[1]: Entering directory `/home/yunq/Pynq-Python/backup/Pynq-Z1/sdk/arduino_g
rove_ledbar/Debug'
Building file: ../src/arduino_grove_ledbar.c
Invoking: MicroBlaze gcc compiler
mb-gcc -Wall -O0 -g3 -c -fmessage-length=0 -MT"src/arduino_grove_ledbar.o" -I../
../bsp_arduino/iop3_mb/include -mlittle-endian -mcpu=v9.5 -mxl-soft-mul -Wl,--no
-relax -MMD -MP -MF"src/arduino_grove_ledbar.d" -MT"src/arduino_grove_ledbar.d"
-o "src/arduino_grove_ledbar.o" "../src/arduino_grove_ledbar.c"
Finished building: ../src/arduino_grove_ledbar.c

Building target: arduino_grove_ledbar.elf
Invoking: MicroBlaze gcc linker
mb-gcc -Wl,-T -Wl,../src/lscript.ld -L../../bsp_arduino/iop3_mb/lib -mlittle-end
ian -mcpu=v9.5 -mxl-soft-mul -Wl,--no-relax -o "arduino_grove_ledbar.elf"  ./src
/arduino_grove_ledbar.o   -Wl,--start-group,-lxil,-lgcc,-lc,--end-group
Finished building target: arduino_grove_ledbar.elf

Invoking: MicroBlaze Print Size
mb-size arduino_grove_ledbar.elf  |tee "arduino_grove_ledbar.elf.size"
   text    data     bss     dec     hex filename
  18592     676    3364   22632    5868 arduino_grove_ledbar.elf
Finished building: arduino_grove_ledbar.elf.size

Invoking: MicroBlaze Bin Gen
mb-objcopy -O binary arduino_grove_ledbar.elf arduino_grove_ledbar.bin
Finished building: arduino_grove_ledbar.bin

make[1]: Leaving directory `/home/yunq/Pynq-Python/backup/Pynq-Z1/sdk/arduino_gr
ove_ledbar/Debug'
Completed Microblaze Projects' Builds
yunq@xsjl24795-29%
```

This will create the Hardware Platform Project (*hw_def*), and the Board Support Package (*bsp*), and then link and build all the application projects.

If you examine the makefile, you can see how the *MBBINS* variable at the top of the makefile is used to compile the application projects. If you want to add your own custom project to the build process, you need to add the project name to the *MBBINS* variable, and save the project in the same location as the other application projects.

Individual projects can be built by navigating to the `<project directory>/Debug` and running `make`.

Binary files

Compiling code produces an executable file (.elf) which needs to be converted to binary format (.bin) to be downloaded to, and run on, an IOP.

A .bin file can be generated from a .elf by running the following command from the SDK shell:

```
mb-objcopy -O binary <inputfile>.elf <outputfile>.bin
```

This is done automatically by the makefile for the existing application projects. The makefile will also copy all .bin files into the `<GitHub Repository>/Pynq-Z1/sdk/bin` folder.

Creating your own Application project

Using the akefile flow, you can use an existing project as a starting point for your own project.

Copy and rename the project, and modify or replace the .c file in the src/ with your C code. The generated .bin file will have the same base name as your C file.

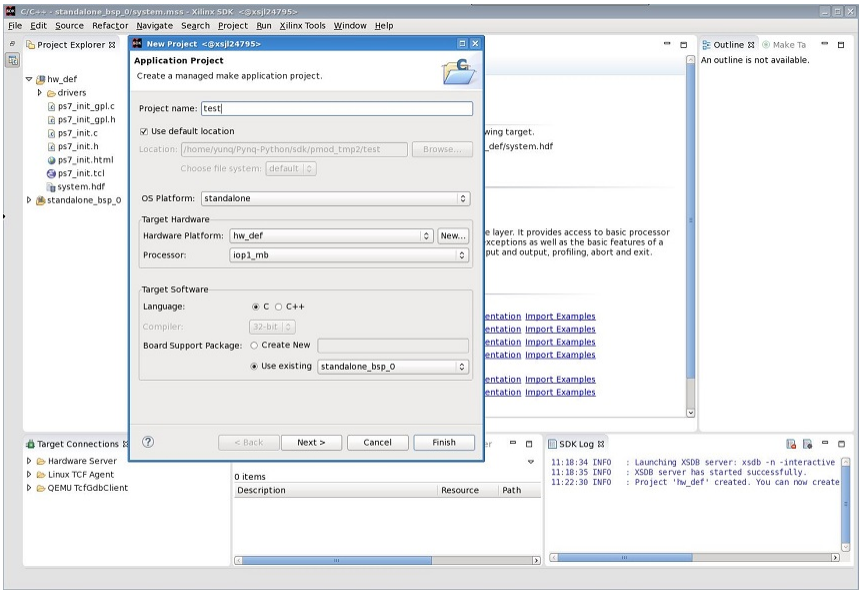
e.g. if your C code is my_peripheral.c, the generated .elf and .bin will be my_peripheral.elf and my_peripheral.bin.

The following naming convention is recommended for peripheral applications <pmod|grove|arduino>_<peripheral>

You will need to update references from the old project name to your new project name in `<project directory>/Debug/makefile` and `<project directory>/Debug/src/subdir.mk`

If you want your project to build in the main makefile, you should also append the .bin name of your project to the *MBBINS* variable at the top of the makefile.

If you are using the SDK GUI, you can import the Hardware Platform, BSP, and any application projects into your SDK workspace.



The SDK GUI can be used to build and debug your code.

IOP Memory architecture

Each IOP has local memory (implemented in Xilinx BRAMs) and a connection to the PS DDR memory.

The IOP instruction and data memory is implemented in a dual port Block RAM, with one port connected to the IOP, and the other to the ARM processor. This allows an executable binary file to be written from the ARM (i.e. the Pynq environment) to the IOP instruction memory. The IOP can also be reset from Pynq, allowing the IOP to start executing the new program.

The IOP data memory, either in local memory, or in DDR memory, can be used as a mailbox for communication and data exchanges between the Pynq environment and the IOP.

DDR memory buffer

DDR memory is managed by the Linux kernel running on the Cortex-A9s. Therefore, the IOP must first be allocated memory regions to access DRAM – this allocation is accomplished within pynq using the xlnk driver.

One benefit of using the pynq xlnk driver is that the physical address is also recorded. By having that mapping, Pynq applications can then send the physical address of that buffer to programmable logic as a pointer.

A single IOP, or multiple IOPs or other devices in an overlay could access this additional memory. For multiple IOPs accessing the same memory buffer, the user should determine a convention to ensure data is not corrupted.

For example, a mailbox could be defined inside a shared memory buffer with specific read and write locations for each IOP. The Python application would need to reserve the required memory buffer for this mailbox.

Shared Memory	IOP1	IOP2
buffer(0)	command (write)	command (read)
buffer(1)	acknowledge(read)	acknowledge(write)
buffer(100->199)	data (write)	data(read)
buffer(200->299)	data (read)	data(write)

Remember that there is no memory protection, and nothing to stop an IOP writing to any location, so these read/write addresses should be managed by the IOP application designer.

DDR memory connect

The IOPs are connected to the DDR memory via the General Purpose AXI slave port. This is a direct connection, and no DMA is available, so is only suitable for simple data transfers from the IOP. I.e. The MicroBlaze can attempt to read or write the DDR as quickly as possible in a loop, but there is no support for bursts, or streaming data.

IOP Memory map

The local IOP memory is 64KB of shared data and instruction memory. Instruction memory for the IOP starts at address 0x0.

Pynq and the application running on the IOP can write to anywhere in the shared memory space. You should be careful not to write to the instruction memory unintentionally as this will corrupt the running application.

When building the MicroBlaze project, the compiler will only ensure that the application and *allocated* stack and heap fit into the BRAM and DDR if used. For communication between the ARM and the MicroBlaze, a part of the shared memory space must also be reserved within the MicroBlaze address space.

There is no memory management in the IOP. You must ensure the application, including stack and heap, do not overflow into the defined data area. Remember that declaring a stack and heap size only allocates space to the stack and heap. No boundary is created, so if sufficient space was not allocated, the stack and heap may overflow and corrupt your application.

If you need to modify the stack and heap for an application, the linker script can be found in the `<project>/src/` directory.

It is recommended to follow the same convention for data communication between the two processors via a MAILBOX.

Instruction and data memory start	0x0
Instruction and data memory size	0xf000
Shared mailbox memory start	0xf000
Shared mailbox memory size	0x1000
Shared mailbox Command Address	0xfffc

These MAILBOX values for an IOP application are defined here:

```
<GitHub Repository>/Pynq-Z1/vivado/ip/arduino_io_switch_1.0/  \
drivers/arduino_io_switch_1.0/src/arduino.h
<GitHub Repository>/Pynq-Z1/vivado/ip/pmod_io_switch_1.0/  \
drivers/pmod_io_switch_1.0/src/pmod.h
```

The corresponding Python constants are defined here:

```
<GitHub Repository>/python/pynq/iop/iop_const.py
```

The following example explains how Python could initiate a read from a peripheral connected to an IOP.

1. Python writes a read command (e.g. 0x3) to the mailbox command address (0xfffc).
2. MicroBlaze application checks the command address, and reads and decodes the command.
3. MicroBlaze performs a read from the peripheral and places the data at the mailbox base address (0xf000).
4. MicroBlaze writes 0x0 to the mailbox command address (0xfffc) to confirm transaction is complete.
5. Python checks the command address (0xfffc), and sees that the MicroBlaze has written 0x0, indicating the read is complete, and data is available.
6. Python reads the data in the mailbox base address (0xf000), completing the read.

Running code on different IOPs

The MicroBlaze local BRAM memory is mapped into the MircoBlaze address space, and also to the ARM address space. These address spaces are independent, so the local memory will be located at different addresses in each memory space. Some example mappings are shown below to highlight the address translation between MicroBlaze and ARM's memory spaces.

IOP Base Address	MicroBlaze Address Space	ARM Equivalent Address Space
0x4000_0000	0x0000_0000 - 0x0000_ffff	0x4000_0000 - 0x4000_ffff
0x4200_0000	0x0000_0000 - 0x0000_ffff	0x4200_0000 - 0x4200_ffff
0x4400_0000	0x0000_0000 - 0x0000_ffff	0x4400_0000 - 0x4400_ffff

Note that each MicroBlaze has the same range for its address space. However, the location of each IOPs address space in the ARM memory map is different for each IOP. As the address space is the same for each IOP, any binary compiled for one Pmod IOP will work on another Pmod IOP.

e.g. if IOP1 exists at 0x4000_0000, and IOP2 (a second instance of an IOP) exists at 0x4200_0000, the same binary can run on IOP1 by writing the binary from python to the 0x4000_0000 address space, and on IOP2 by writing to the 0x4200_0000.

CHAPTER 14

IO Processors: Writing applications

Table of Contents

Introduction

The previous section described the software architecture and the software build process. This section will cover how to write the IOP application and also the corresponding Python interface.

The section assumes that the hardware platform and the BSPs have already been generated as detailed in the previous section.

IOP header files and libraries

A library is provided for the IOPs which includes an API for local peripherals (IIC, SPI, Timer, Uart, GPIO), the configurable switch, links to the peripheral addresses, and mappings for the mailbox used in the existing IOP peripheral applications provided with Pynq. This library can be used to write custom IOP applications.

The only IP that is specific to each IOP is the configurable switch. There is a `pmod_io_switch` and an `arduino_io_switch`. The header files for the IOPs are associated with the corresponding configurable switch, and can be found here

```
<GitHub Repository>/Pynq-Z1/vivado/ip/pmod_io_switch_1.0/  \
drivers/pmod_io_switch_v1_0/src/pmod.h
   <GitHub Repository>/Pynq-Z1/vivado/ip/arduino_io_switch_1.0/  \
drivers/arduino_io_switch_v1_0/src/arduino.h
```

The corresponding C code, `pmod.c` and `arduino.c` can also be found in this directory.

Configurable switch header files

There is a separate header file that corresponds to each configurable switch. These files include the API for the configuration switch and predefined constants that can be used to connect to the physical interface on the board.

Pmod Configurable Switch header

You can find the header file for the Pmod IOP switch here:

```
<GitHub Repository>/Pynq-Z1/vivado/ip/pmod_io_switch_1.0/  \
drivers/pmod_io_switch_v1_0/src/pmod_io_switch.h
```

This code is automatically compiled into the Board Support Package (BSP).

Arduino

The corresponding files for the Arduino IOP switch can be found here:

```
<GitHub Repository>/Pynq-Z1/vivado/ip/arduino_io_switch_1.0/  \
drivers/arduino_io_switch_1.0/src/arduino_io_switch.h
```

Files to include

To use these files in an IOP application, include the header file(s):

For a Pmod IOP:

```
#include "pmod.h"
#include "pmod_io_switch.h"
```

or for an Arduino IOP:

```
#include "arduino.h"
#include "arduino_io_switch.h"
```

Pmod applications should call `pmod_init()` at the beginning of the application, and Arduino applications, `arduino_init()`. This will initialize all the IOP peripherals in the subsystem.

Controlling the Pmod IOP Switch

The IOP switch needs to be configured by the IOP application before any peripherals can be used. This can be done statically from within the application, or the application can allow Python to write a switch configuration to shared memory, which can be used to configure the switch. This functionality must be implemented by the user, but existing IOP applications can be used as a guide. For example, the `arduino_lcd18` IOP project shows and example of reading the switch configuration from the mailbox, and using this to configure the switch.

There are 8 data pins on a Pmod port, that can be connected to any of 16 internal peripheral pins (8x GPIO, 2x SPI, 4x IIC, 2x Timer). This means the configuration switch for the Pmod has 8 connections to make to the data pins.

Each pin can be configured by writing a 4 bit value to the corresponding place in the IOP Switch configuration register. The first nibble (4-bits) configures the first pin, the second nibble the second pin and so on.

The following function, part of the provided pmod_io_switch_v1_0 driver (`pmod.h`) can be used to configure the switch from an IOP application.

```
void config_pmod_switch(char pin0, char pin1, char pin2, char pin3, char pin4, \
    char pin5, char pin6, char pin7);
```

While each parameter is a "char" only the lower 4-bits are used to configure each pin.

Switch mappings used for IOP Switch configuration:

Pin	Value
GPIO_0	0x0
GPIO_1	0x1
GPIO_2	0x2
GPIO_3	0x3
GPIO_4	0x4
GPIO_5	0x5
GPIO_6	0x6
GPIO_7	0x7
SCL	0x8
SDA	0x9
SPICLK	0xa
MISO	0xb
MOSI	0xc
SS	0xd
PWM	0xe
TIMER	0xf

Example

```
config_pmod_switch(SS,MOSI,GPIO_2,SPICLK,GPIO_4,GPIO_5,GPIO_6,GPIO_7);
```

This would connect a SPI interface:

- Pin 0: SS
- Pin 1: MOSI
- Pin 2: GPIO_2
- Pin 3: SPICLK
- Pin 4: GPIO_4
- Pin 5: GPIO_5
- Pin 6: GPIO_6
- Pin 7: GPIO_7

Note that if two or more pins are connected to the same signal, the pins are OR'd together internally.

```
config_pmod_switch(GPIO_1,GPIO_1,GPIO_1,GPIO_1,GPIO_1,GPIO_1,GPIO_1,GPIO_1);
```

This is not recommended and should not be done unintentionally.

Controlling the Arduino IOP Switch

Switch mappings used for IO switch configuration:

Pin	A/D IO	A_INT	Inter-rupt	UART	PWM	Timer	SPI	IIC	Input-Capture
A0	A_GPIO	A_INT							
A1	A_GPIO	A_INT							
A2	A_GPIO	A_INT							
A3	A_GPIO	A_INT							
A4	A_GPIO	A_INT						IIC	
A5	A_GPIO	A_INT						IIC	
D0	D_GPIO		D_INT	D_UART					
D1	D_GPIO		D_INT	D_UART					
D2	D_GPIO		D_INT						
D3	D_GPIO		D_INT		D_PWM0	D_TIMER Timer0			IC Timer0
D4	D_GPIO		D_INT			D_TIMER Timer0_6			
D5	D_GPIO		D_INT		D_PWM1	D_TIMER Timer1			IC Timer1
D6	D_GPIO		D_INT		D_PWM2	D_TIMER Timer2			IC Timer2
D7	D_GPIO		D_INT						
D8	D_GPIO		D_INT			D_TIMER Timer1_7			Input Capture
D9	D_GPIO		D_INT		D_PWM3	D_TIMER Timer3			IC Timer3
D10	D_GPIO		D_INT		D_PWM4	D_TIMER Timer4	D_SS		IC Timer4
D11	D_GPIO		D_INT		D_PWM5	D_TIMER Timer5	D_MOSI		IC Timer5
D12	D_GPIO		D_INT				D_MISO		
D13	D_GPIO		D_INT				D_SPICLK		

For example, to connect the UART to D0 and D1, write D_UART to the configuration register for D0 and D1.

```
config_arduino_switch(A_GPIO, A_GPIO, A_GPIO, A_GPIO, A_GPIO, A_GPIO,
                      D_UART, D_UART, D_GPIO, D_GPIO, D_GPIO,
                      D_GPIO, D_GPIO, D_GPIO, D_GPIO,
                      D_GPIO, D_GPIO, D_GPIO, D_GPIO);
```

IOP Application Example

Taking Pmod ALS as an example IOP driver (used to control the PMOD light sensor):

`<GitHub Repository>/Pynq-Z1/sdk/pmod_als/src/pmod_als.c`

First note that the `pmod.h` header file is included.

```
#include "pmod.h"
```

Some *COMMANDS* are defined. These values can be chosen to be any value. The corresponding Python code will send the appropriate command values to control the IOP application.

By convention, 0x0 is reserved for no command/idle/acknowledge, and IOP commands can be any non-zero value.

```
// MAILBOX_WRITE_CMD
#define READ_SINGLE_VALUE 0x3
```

```
#define READ_AND_LOG      0x7
// Log constants
#define LOG_BASE_ADDRESS (MAILBOX_DATA_PTR(4))
#define LOG_ITEM_SIZE sizeof(u32)
#define LOG_CAPACITY  (4000/LOG_ITEM_SIZE)
```

The ALS peripheral has as SPI interface. The user defined function get_sample() calls an SPI function *spi_transfer()*, defined in pmod.h, to read data from the device.

```
u32 get_sample(){
   /*
   ALS data is 8-bit in the middle of 16-bit stream.
   Two bytes need to be read, and data extracted.
   */
   u8 raw_data[2];
   spi_transfer(SPI_BASEADDR, 2, raw_data, NULL);
   //  return ( ((raw_data[0] & 0xf0) >> 4) + ((raw_data[1] & 0x0f) << 4) );
   return ( ((raw_data[1] & 0xf0) >> 4) + ((raw_data[0] & 0x0f) << 4) );
}
```

In `main()` notice `config_pmod_switch()` is called to initialize the switch with a static configuration. This application does not allow the switch configuration to be modified from Python. This means that if you want to use this code with a different pin configuration, the C code must be modified and recompiled.

```
int main(void)
{
   int cmd;
   u16 als_data;
   u32 delay;

   pmod_init(0,1);
   config_pmod_switch(SS, GPIO_1, MISO, SPICLK, \
                      GPIO_4, GPIO_5, GPIO_6, GPIO_7);
   // to initialize the device
   get_sample();
```

Next, the `while(1)` loop continually checks the `MAILBOX_CMD_ADDR` for a non-zero command. Once a command is received from Python, the command is decoded, and executed.

```
// Run application
while(1){

   // wait and store valid command
   while((MAILBOX_CMD_ADDR & 0x01)==0);
   cmd = MAILBOX_CMD_ADDR;
```

Taking the first case, reading a single value; `get_sample()` is called and a value returned to the first position (0) of the `MAILBOX_DATA`.

`MAILBOX_CMD_ADDR` is reset to zero to acknowledge to the ARM processor that the operation is complete and data is available in the mailbox.

```
switch(cmd){
   case READ_SINGLE_VALUE:
   // write out reading, reset mailbox
   MAILBOX_DATA(0) = get_sample();
   MAILBOX_CMD_ADDR = 0x0;
   break;
```

Remaining code:

```
            case READ_AND_LOG:
            // initialize logging variables, reset cmd
            cb_init(&pmod_log, LOG_BASE_ADDRESS, LOG_CAPACITY, LOG_ITEM_SIZE);
            delay = MAILBOX_DATA(1);
            MAILBOX_CMD_ADDR = 0x0;

               do{
                  als_data = get_sample();
                  cb_push_back(&pmod_log, &als_data);
                  delay_ms(delay);
               } while((MAILBOX_CMD_ADDR & 0x1)== 0);

               break;

            default:
               // reset command
               MAILBOX_CMD_ADDR = 0x0;
               break;
         }
      }
      return(0);
   }
```

Examining the Python Code

With the IOP Driver written, the Python class can be built that will communicate with that IOP.

`<GitHub Repository>/python/pynq/iop/pmod_als.py`

First the MMIO, request_iop, iop_const, PMODA and PMODB are imported.

```
import time
from pynq import MMIO
from pynq.iop import request_iop
from pynq.iop import iop_const
from pynq.iop import PMODA
from pynq.iop import PMODB

ALS_PROGRAM = "pmod_als.bin"
```

The MicroBlaze binary for the IOP is also declared. This is the application executable, and will be loaded into the IOP instruction memory.

The ALS class and an initialization method are defined:

```
class Pmod_ALS(object):
   def __init__(self, if_id):
```

The initialization function for the module requires an IOP index. For Grove peripherals and the StickIt connector, the StickIt port number can also be used for initialization. The `__init__` is called when a module is instantiated. e.g. from Python:

```
from pynq.pmods import Pmod_ALS
als = Pmod_ALS(PMODB)
```

Looking further into the initialization method, the `_iop.request_iop()` call instantiates an instance of an IOP on the specified pmod_id and loads the MicroBlaze executable (ALS_PROGRAM) into the instruction memory of the appropriate MicroBlaze.

```
self.iop = request_iop(if_id, PMOD_ALS_PROGRAM)
```

An MMIO class is also instantiated to enable read and write to the shared memory.

```
self.mmio = self.iop.mmio
```

Finally, the iop.start() call pulls the IOP out of reset. After this, the IOP will be running the als.bin executable.

```
self.iop.start()
```

Example of Python Class Runtime Methods

The read method in the Pmod_ALS class will simply read an ALS sample and return that value to the caller. The following steps demonstrate a Python to MicroBlaze read transaction specific to the ALS class.

```
def read(self):
```

First, the command is written to the MicroBlaze shared memory using mmio.write(). In this case the value 0x3 represents a read command. This value is user defined in the Python code, and must match the value the C program running on the IOP expects for the same function.

```
self.mmio.write(iop_const.MAILBOX_OFFSET+
                    iop_const.MAILBOX_PY2IOP_CMD_OFFSET, 3)
```

When the IOP is finished, it will write 0x0 to the command area. The Python code now uses mmio.read() to check if the command is still pending (in this case, when the 0x3 value is still present at the `CMD_OFFSET`). While the command is pending, the Python class blocks.

```
while (self.mmio.read(iop_const.MAILBOX_OFFSET+
                              iop_const.MAILBOX_PY2IOP_CMD_OFFSET) == 3):
    pass
```

Once the command is no longer 0x3, i.e. the acknowledge has been received, the result is read from the `DATA` area of the shared memory `MAILBOX_OFFSET` using *mmio.read()*.

```
return self.mmio.read(iop_const.MAILBOX_OFFSET)
```

Notice the iop_const values are used in these function calls, values that are predefined in `iop_const.py`.

CHAPTER 15

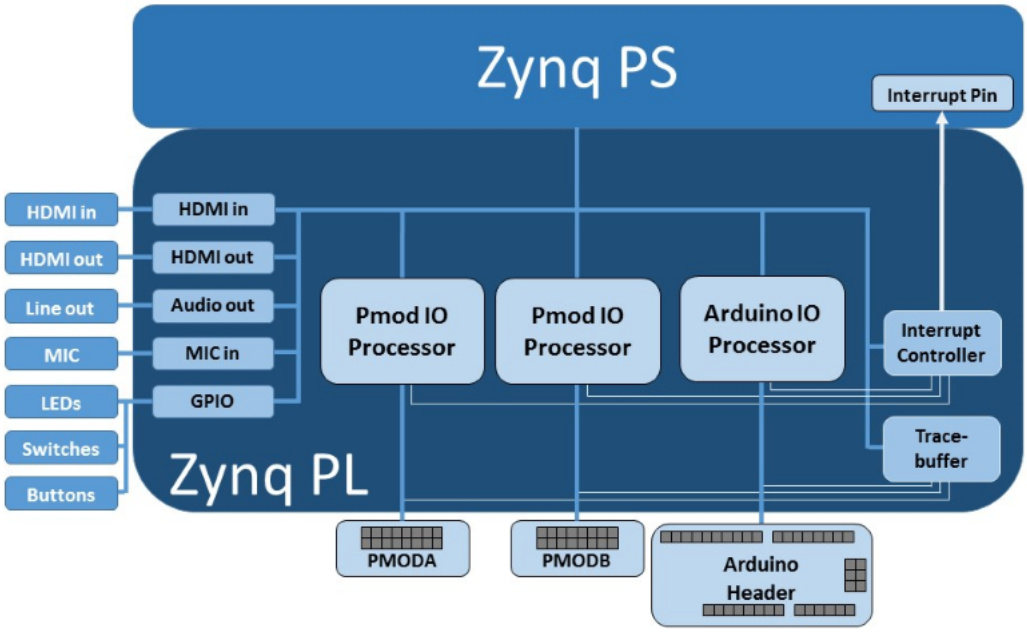
Interrupts

Table of Contents

Introduction

Each IOP has its only interrupt controller. This allows IOP peripherals (IIC, SPI, GPIO, Uart, Timers) to interrupt the MicroBlaze processor inside the IOP. The IOP uses the AXI Interrupt Controller. It can be used in an IOP application in the same way as any other MicroBlaze application to manage this local interrupts.

The base overlay also has a interrupt controller connected to the interrupt pin of the Zynq PS. The overlay interrupt controller can be triggered by the MicroBlaze inside an IOP to signal to the PS and Python that an interrupt has occurred in the overlay.



Interrupts in PYNQ can be handled in different ways. The *asyncio* Python package is one method of handling interrupts. Asyncio was first introduced in Python 3.4 as provisional, and starting in Python 3.6 is considered stable. Python 3.6 documentation on asyncio.

This PYNQ release used Python 3.6 and includes the latest asyncio package.

The main advantage of using asyncio over other interrupt handling methods, is that it makes the interrupt handler look similar to regular Python code. This helps reduce the complexity of managing interrupts using callbacks.

It should be noted that Python is a productivity language rather than a performance language. Any performance critical, or real-time parts of a design should be handled in the PL. An interrupt sent to the PS may have a relatively long latency before it is handled.

Asyncio

Background terminology

Asyncio includes the following components:

Event loop

An event loop is a loop for scheduling multiple asynchronous functions. When an event loop runs, and the first IO function is reached, the function pauses waiting for its IO to complete. While the function is waiting, the loop continues, executing subsequent functions in the same way. When a function completes its IO, it can resume at the next scheduled point in the event loop.

```
loop = asyncio.get_event_loop()
```

https://docs.python.org/3/library/asyncio-eventloop.html

Futures

A future is an object that will have a value in the future. The event loop can wait for a *Future* object to be set to done. i.e. data available.

```
asyncio.ensure_future(async_coroutine(5)),
```

https://docs.python.org/3/library/asyncio-task.html#future

Coroutines

A coroutine is a function that can pause, that can receive values, and can return a series of value periodically. A coroutine is a functions decorated with `async def` (Python 3.6).

```
async def function():
    ...
```

https://docs.python.org/3/library/asyncio-task.html#coroutines

Tasks

A task is a coroutine wrapped inside a Future. A task runs as long as the event loop runs.

```
asyncio.ensure_future(async_coroutine())
```

https://docs.python.org/3/library/asyncio-task.html#task

await

The `await` expression is used to obtain a result from a coroutine

```
async def asyncio_function(db):
    data = await read()
    ...
```

Example

An event loop registers a task object. The loop will schedule and run the task. Callbacks can be added to the task to notify when a future has a result.

When the coroutine in a task *awaits* it is paused. When it has a value, it resumes. When it returns, the task completes, and the future gets a value. Any associated callback is run.

```
async def async_coroutine(max):
   for i in range (1,max):
      await asyncio.sleep(1)
      print(i)

   print("Done")

loop = asyncio.get_event_loop()
tasks = [
   asyncio.ensure_future(async_coroutine(5)),
   asyncio.ensure_future(async_coroutine(20)),
   asyncio.ensure_future(async_coroutine(10)),
```

```
   asyncio.ensure_future(async_coroutine(1))]
loop.run_until_complete(asyncio.gather(*tasks))
loop.close()
```

Asyncio requirements

All blocking calls in event loop should be replaced with coroutines. If you do not do this, when a blocking call is reached, it will block the rest of the loop.

If you need blocking calls, they should be in separate threads.

Compute workloads should be in separate threads/processes.

Interrupts in PYNQ using asyncio

Asyncio can be used for managing interrupt events from the overlay. A coroutine can be run in an event loop and used to check the status of the interrupt controller in the overlay, and handle any event. Other user functions can also be run in the event loop. If an interrupt is triggered, the next time the "interrupt" coroutine is scheduled, it will service the interrupt. The responsiveness of the interrupt coroutine will depend on how frequently the user code yields control in the loop.

Interrupts in the Base Overlay

The I/O peripherals in the base overlay will trigger interrupts when switches are toggled or buttons are pressed. Both the *Button* and *Switch* classes have a function `wait_for_level` and a coroutine `wait_for_level_async` which block until the corresponding button or switch has the specified value. This follows a convention throughout the PYNQ python API that that coroutines have an `_async` suffix.

As an example, consider an application where each LED will light up when the corresponding button is pressed. First a coroutine specifying this functionality is defined:

```
async def button_to_led(number):
    button = pynq.board.Button(number)
    led = pynq.board.LED(number)
    while True:
        await button.wait_for_level_async(1)
        led.on()
        await button.wait_for_level_async(0)
        led.off()
```

Next add instances of the coroutine to the default event loop

```
tasks = [asyncio.ensure_future(button_to_led(i) for i in range(4)]
```

Finally, running the event loop will cause the coroutines to be active. This code runs the event loop until an exception is thrown or the user interrupts the process.

```
asyncio.get_event_loop().run_forever()
```

IOP and Interrupts

The IOP class has an `interrupt` member variable which acts like an *asyncio.Event* with a `wait` coroutine and a `clear` method. This event is automatically wired to the correct interrupt pin or set to `None` if interrupts are not

available in the loaded overlay.

e.g.

```
def __init__(self)
    self.iop = request_iop(iop_id, IOP_EXECUTABLE)
    if self.iop.interrupt is None:
       warn("Interrupts not available in this Overlay")
```

There are two options for running functions from this new IOP wrapper class. The function can be called from an external asyncio event loop (set up elsewhere), or the function can set up its own event loop and then call its asyncio function from the event loop.

Async function

By convention, the PYNQ python API offers both an asyncio coroutine and a blocking function call for all interrupt-driven functions. It is recommended that this should be extended to any user-provided IOP drivers. The blocking function can be used where there is no need to work with asyncio, or as a convenience function to run the event loop until a specified condition. The coroutine is given the `_async` suffix to avoid breaking backwards compatibility when updating existing functions.

The following code defines an asyncio coroutine. Notice the `async` and `await` keywords are the only additional code needed to make this function an asyncio coroutine.

```
async def interrupt_handler_async(self, value):
    if self.iop.interrupt is None:
        raise RuntimeError('Interrupts not available in this Overlay')
    while(1):
        await self.iop.interrupt.wait() # Wait for interrupt
        # Do something when an interrupt is received
        self.iop.interrupt.clear()
```

Function with event loop

The following code wraps the asyncio coroutine, adding to the default event loop and running it until the coroutine completes.

```
def interrupt_handler(self):

    if self.interrupt is None:
        raise RuntimeError('Interrupts not available in this Overlay')
    loop = asyncio.get_event_loop()
    loop.run_until_complete(asyncio.ensure_future(
        self.interrupt_handler_async()
    ))
```

Custom interrupt handling

The Python *Interrupt* class can be found here:

```
<GitHub Repository>\pynq\interrupt.py
```

This class abstracts away management of the AXI interrupt controller in the PL. It is not necessary to examine this code in detail to use interrupts. The interrupt class takes the pin name of the interrupt line and offers a single `wait`

coroutine. The interrupt is only enabled in the hardware for as long as a coroutine is waiting on an *Interrupt* object. The general pattern for using an Interrupt is as follows:

```
while condition:
    await interrupt.wait()
    # Clear interrupt
```

This pattern avoids race conditions between the interrupt and the controller and ensures that an interrupt isn't seen multiple times.

Interrupt pin mappings

Interrupts are also available from the GPIO (Pushbuttons, Switches, Video, Trace buffer Arduino, Trace buffer Pmods).

Name	IOP ID	Pin
PMODA	1	iop1/dff_en_reset_0/q
PMODB	2	iop2/dff_en_reset_0/q
ARDUINO	3	iop3/dff_en_reset_0/q
Buttons		btns_gpio/ip2intc_irpt
Switches		swsleds_gpio/ip2intc_irpt
Video		video/dout
Trace(Pmod)		tracepmods_arduino/s2mm_introut
Trace(Arduino)		tracebuffer_arduino/s2mm_introut

Interrupt examples using asyncio

Example notebooks

The asyncio_buttons.ipynb notebook can be found in the examples directory. The Arduino LCD IOP driver provides an example of using the IOP interrupts.

CHAPTER 16

Creating Overlays

Table of Contents

Introduction

As described in the PYNQ introduction, overlays are analogous to software libraries. A programmer can download overlays into the Zynq® PL at runtime to provide functionality required by the software application.

An *overlay* is a class of Programmable Logic design. Programmable Logic designs are usually highly optimized for a specific task. Overlays however, are designed to be configurable, and reusable for broad set of applications. A PYNQ overlay will have a Python interface, allowing a software programmer to use it like any other Python package.

A software programmer can use an overlay, but will not usually create overlay, as this usually requires a high degree of hardware design expertise.

This section will give an overview of the process of creating an overlay and integrating it into PYNQ, but will not cover the hardware design process in detail. Hardware design will be familiar to Zynq, and FPGA hardware developers.

Vivado design

An overlay consists of two main parts; the Programmable Logic (PL) design, and the Python API.

Xilinx® Vivado software is used to create the PL design. This will generate a *bitstream* or *binary* file (.bit file) that is used to program the Zynq PL.

The free WebPack version of Vivado can be used with the PYNQ-Z1 board to create overlays. https://www.xilinx.com/products/design-tools/vivado/vivado-webpack.html

There are some differences between the standard Zynq design process, and designing overlays for PYNQ. A Vivado project for a Zynq design consists of two parts; the PL design, and the PS configuration settings. The PS configuration includes settings for system clocks, including the clocks used in the PL.

The PYNQ image which is used to boot the board configures the Zynq PS at boot time. Overlays are downloaded as required by the programmer, and will not reconfigure the Zynq PS. This means that overlay designers should ensure the PS settings in their Vivado project match the PYNQ image settings.

The following settings should be used for a new Vivado overlay project:

Vivado Project settings:

- Target device: xc7z020clg400-1

PL clock configuration:

- FCLK_CLK0: 100.00MHz
- FCLK_CLK1: 142.86MHz
- FCLK_CLK2: 200.00MHz
- FCLK_CLK3: 166.67MHz

The PYNQ-Z1 Master XDC (I/O constraints) are available at the Digilent PYNQ-Z1 resource site: https://reference.digilentinc.com/reference/programmable-logic/pynq-z1/start

It is recommended to start with an existing overlay design to ensure the PS settings are correct. The source files for the *base* overlay can be found in the pynq GitHub, and the project can be rebuilt using the makefile available here:

```
<GitHub repository>/Pynq-Z1/vivado/base
```

Block Diagram Tcl

The tcl for the Vivado block diagram should also be exported with the bitstream. This allows information about the overlay to be parsed into Python (e.g. list of IPs in the overlay). See the next section for details on how to query the tcl file.

You can use a custom tcl file to build your project, or block diagram, but custom tcl files may not be parsed correctly. You should use Vivado to export the tcl for the block diagram. This should ensure it can be parsed correctly in Python.

To generate the tcl for the Block Diagram from the Vivado GUI:

- Click **File > Export > Block Design**

Or, run the following in the tcl console:

```
write_bd_tcl
```

The tcl filename should match the .bit filename. E.g. my_overlay.bit and my_overlay.tcl

The tcl is parsed when the overlay is instantiated (not when it is downloaded).

```
from pynq import Overlay
ol = Overlay("base.bit") # tcl is parsed here
```

An error will be displayed if a tcl is not available when attempting to download an overlay, or if the tcl filename does not match the .bit file name.

ip_dict

The Overlay package generates a dictionary called ip_dict containing the names of IP in a specific overlay (e.g. *base.bit*). The dictionary can be used to reference an IP by name in your Python code, rather than by a hard coded address. It can also check the IP available in an overlay.

To show the IP dictionary of the overlay, run the following:

```
from pynq import Overlay
OL = Overlay("base.bit")
OL.ip_dict
```

Each entry in this IP dictionary that is returned is a key-value pair.

E.g.:

```
'SEG_axi_dma_0_Reg':  [2151677952, 65536, None],
```

Note, this parses the tcl file that was exported with the bitstream. It does not do check the overlay currently running in the PL.

The key of the entry is the IP instance name; all the IP instance names are parsed from the **.tcl* file (e.g. *base.tcl*) in the address segment section. The value of the entry is a list of 3 items:

- The first item shows the base address of the addressable IP (as an int).
- The second item shows the address range in bytes (as an int).
- The third item records the state associated with the IP. It is *None* by default, but can be user defined.

Similarly, the PL package can be used to find the addressable IPs currently in the programmable logic:

```
from pynq import PL
PL.ip_dict
```

Existing Overlays

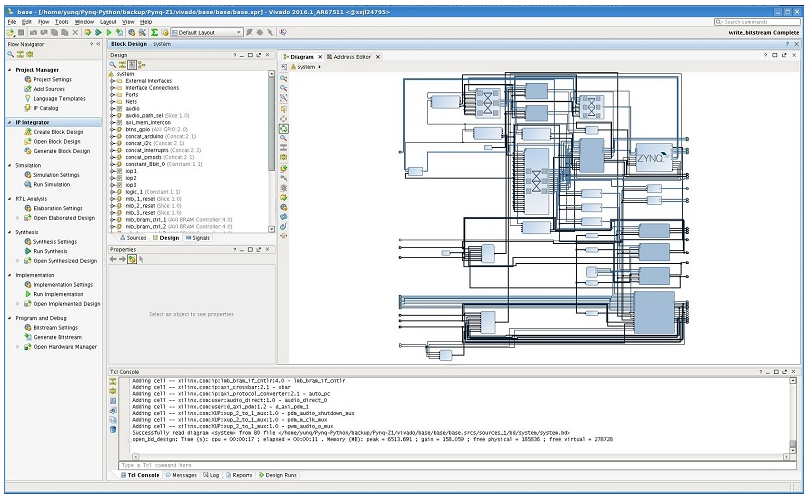
The *base* overlay is included in the Pynq repository and can be found here:

```
<GitHub repository>/Pynq-Z1/vivado/base
```

A makefile exists in each folder that can be used to rebuild the Vivado project and generate the bitstream for the overlay. The bitstream and tcl for the overlay are available on the board (base.bit is loaded by default when the board boots), and in the project repository:

```
<GitHub Repository>/Pynq-Z1/bitstream/
```

Vivado must be installed to design and build overlays. Building an existing overlay design allows the project to be opened in Vivado and examined, or modified to create a new overlay.



Interfacing to an overlay

MMIO

PYNQ includes the *MMIO* Python class to simplify communication between the Zynq PS and PL. Once the overlay has been created, and the memory map is known, the *MMIO* can be used to access memory mapped locations in the PL.

The Python code for the MMIO can be viewed here:

```
<GitHub Repository>/python/pynq/mmio.py
```

The MMIO class can access an area of memory in the PL by specifying the start address, and the range. E.g. The following code allows access to memory mapped locations in the PL from 0x40000000 to 0x40010000 (*SEG_mb_bram_ctrl_1_Mem0*):

```
from pynq import MMIO

# an IP is located at 0x40000000
myip = MMIO(0x40000000,0x10000)

# Read from the IP at offset 0
myip.read(0)
```

In the example above, any accesses outside the address range 0x10000 (65535 bytes) will cause an exception in the MMIO package. The designer must also be careful to ensure that addresses accessed by the MMIO have something mapped in the PL. Remember that custom peripherals exist in the address space, and even if and address range is mapped by the MMIO, there may not be anything connected to specific addresses, or they may be read only or write only. Invalid accesses to the PL will cause system errors and will likely crash a Jupyter kernel.

When creating the python driver for a new hardware function, the MMIO can be wrapped inside a Python module.

Zynq GPIOs

GPIO between the Zynq PS and PL can be used by Python code as a control interface to overlays. The information about a GPIO is kept in the GPIO dictionary of an overlay, similar to the *ip_dict* discussed above.

The following code can be used to get the dictionary for a bitstream:

```
from pynq import Overlay
ol = Overlay("base.bit")
ol.gpio_dict
```

A GPIO dictionary entry is a key, value pair, where *value* is a list of two items. An example of the entry in a GPIO dictionary:

```
'mb_1_reset/Din':  [0, None]
```

The key is the GPIO instance name (*mb_1_reset/Din*). GPIO instance names are read and parsed from the Vivado **.tcl* file (e.g. *base.tcl*).

The *value* is a list of 2 items:

- The first item shows the index of the GPIO (0).
- The second item (*None*) shows the state of the GPIO. It is *None* by default, but can be user defined.

The following code can be used to get the dictionary for GPIO currently in the FPGA fabric:

```
from pynq import PL
pl = PL
pl.gpio_dict
```

CFFI

CFFI (C Foreign Function Interface) provides a simple way to interface with C code from Python. The CFFI package is preinstalled in the PYNQ image. It supports an inline ABI (Application Binary Interface) compatibility mode, which allows you to dynamically load and run functions from executable modules, and an API mode, which allows you to build C extension modules.

The following example taken from http://docs.python-guide.org/en/latest/scenarios/clibs/ shows the ABI inline mode, calling the C function `strlen()` in from Python

C function prototype:

```
size_t strlen(const char*);
```

The C function prototype is passed to `cdef()`, and can be called using `clib`.

```
from cffi import FFI
ffi = FFI()
ffi.cdef("size_t strlen(const char*);")
clib = ffi.dlopen(None)
length = clib.strlen(b"String to be evaluated.")
print("{}".format(length))
```

C functions inside a shared library can be called from Python using the C Foreign Function Interface (CFFI). The shared library can be compiled online using the CFFI from Python, or it can be compiled offline.

For more information on CFFI and shared libraries refer to:

http://cffi.readthedocs.io/en/latest/overview.html

http://www.tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html

To see examples in PYNQ on how to use CFFI, refer to the CMA class or the Audio class, both located:

```
<GitHub Repository>/pynq/drivers
```

Packaging overlays

An overlay, tcl, and Python can be placed anywhere in the filesystem, but this is not good practice.

The default location for the base PYNQ overlay and tcl is :

```
<GitHub Repository>/Pynq-Z1/bitstream
```

The PYNQ Python can be found here:

```
<GitHub Repository>/python/pynq
```

You can fork PYNQ from github, and add Python code to the PYNQ package. However, for custom overlays, you can create your own repository and package it to allow other users to install your overlay using pip.

There are different ways to package a project for installation with pip. One example is provided below.

See pip install for more details, and more packaging options. https://pip.pypa.io/en/stable/reference/pip_install

Example

The following example assume an overlay that exists in the root of a GitHub repository.

Assume the repository has the following structure:

- **notebook/**
  - new_overlay.ipynb
- **new_overlay/**
  - new_overlay.bit
  - new_overlay.tcl
  - __init.py
  - new_overlay.py
- readme.md
- license

Add a setup.py to the root of your repository. This file will imports the necessary packages, and specifies some setup instructions for your package including the package name, version, url, and files to include.

Example setup.py :

```
from setuptools import setup, find_packages
import subprocess
import sys
import shutil
import new_overlay

setup(
    name = "new_overlay",
    version = new_overlay.__version__,
    url = 'https://github.com/your_github/new_overlay',
    license = 'All rights reserved.',
    author = "Your Name",
    author_email = "your@email.com",
    packages = ['new_overlay'],
    package_data = {
```

```
    '' : ['*.bit','*.tcl','*.py','*.so'],
    },
    description = "New custom overlay for PYNQ-Z1"
)
```

package_data specifies which files will be installed as part of the package.

From a terminal, the new package can be installed by running:

```
sudo pip install --upgrade 'git+https://github.com/your_github/new_overlay'
```

Using Overlays

The PL can be dynamically reconfigured with new overlays as the system is running.

Loading overlays can be done in Python using the Overlay class:

```
<GitHub Repository>/python/pynq/pl.py
```

The bitstream can then be downloaded from Python:

```
from pynq import Overlay
ol = Overlay("base.bit")
ol.download()
```

CHAPTER 17

pynq Package

Table of Contents

This section describes the pynq package for the PYNQ-Z1 platform.

After powering the board with the Micro SD card in place, the board will boot into Linux. Python3 is installed with Jupyter Notebook support. The Python package *pynq* allows users to access overlays from Python.

Some pre-installed features of this Linux image include:

- Networking is enabled, and the board will attempt to get an IP address from a DHCP server on the network. If a DHCP server is not found, the board will fallback and assign itself a static IP of 192.168.2.99 by default. This default IP address can be changed.
- Samba, a file sharing service, is enabled. This means that the Linux home area can be accessed from a Windows machine by navigating to or mapping \\pynq\xilinx (Windows) or smb:pynq/xilinx (Mac/Linux) . The samba username:password is xilinx:xilinx. Files can be copied to and from the board, and the Pynq source code, and notebooks can be accessed and modified using your preferred editors on your host PC.
- A Jupyter Notebook server is initialized on port 9090 and automatically starts after boot.
- The base overlay is preloaded in the FPGA fabric.

Python pynq Package Structure

All Pynq code is contained in the *pynq* Python package and is can be found on the board at /home/xilinx/pynq. This package is derived from the Github repository and the latest version can always be installed from <GitHub repository>/python/pynq.

Pynq contains four main subpackages: board, iop, drivers, and bitstream; a tests subpackage is available for testing the user subpackages. Each of the five subpackages are described below.

To learn more about Python package structures, please refer to the official python documentation.

board

This folder contains libraries or python packages for peripherals available on the PYNQ-Z1 board: button, switch, rgbled and led. For example the following code will turn on one of PYNQ-Z1's LEDs:

```
from pynq.board import LED
led = LED(0)
led.on()
```

iop

This folder contains libraries for Pmod devices and Grove peripherals.

`Arduino_Analog`, and `Arduino_IO` are provided for interfacing to the arduino interface.

In addition, `Pmod_IO`, `Pmod_IIC` and `DevMode` are developer classes allowing direct low level access to I/O controllers.

There is also an additional module named `_iop.py`; this module acts like a wrapper to allow all the Pmod classes to interface with the MicroBlaze inside an IOP. The `_IOP` class prevents multiple device instantiations on the same Pmod at the same time. (i.e. This prevents the user assigning more devices to a port than can be physically connected.) `_IOP` uses the overlay class to track each IOP's status.

Note: `_iop.py` is an internal module, not intended to be instantiated directly use. In Python, there is no concept of _public_ and _private_; we use _ as a prefix to indicate internal definitions, variables, functions, and packages.

For example, the following code will instantiate and write to the Pmod_OLED attached on PMODA.

```
from pynq import Overlay
from pynq.iop import Pmod_OLED
from pynq.iop.iop_const import PMODA

ol = Overlay("base.bit")
ol.download()

pmod_oled = Pmod_OLED(PMODA)
pmod_oled.clear()
pmod_oled.write('Welcome to the\nPynq-Z1 board!')
```

bitstream

This folder contains the base.bit and the base.tcl. The base.bit is the precompiled overlay and base.tcl provides information about the hardware it is built from. The tcl file can be parsed by Pynq packages to determine information about the hardware. `PL.ip_dict` will parse the tcl for an overlay and respond with the addresses of any peripherals in the overlay.

drivers

This folder contains various classes to support audio, video, DMA, and Trace_Buffer.

tests

This folder includes a tests package for use with all other pynq subpackages. All testing is done using pytest. Please see The Verification Section to learn more about Pynq's use of pytest to do automated testing.

Note: The `tests` folders in `board`, `iop`, `drivers`, and others rely on the functions implemented in the `test` folders of the pynq package. This common practice in Python where each subpackage has its own `tests`. This practice can keep the source code modular and *self-contained*.

---

documentation

To find documentation for each module, see the Pynq Package for documentation built from the actual Python source code.

CHAPTER 18

pynq package reference

Subpackages

pynq.board package

Submodules

pynq.board.button module

class pynq.board.button.**Button**(*index*)
Bases: object

This class controls the onboard push-buttons.

index
int

Index of the push-buttons, starting from 0.

read()
Read the current value of the button.

Returns Either 1 if the button is pressed or 0 otherwise

Return type int

wait_for_value(*value*)
Wait for the button to be pressed or released

Parameters

- **value** (*int*) – 1 to wait for press or 0 to wait for release
- **function wraps the coroutine form so the asyncio**(*This*) –
- **loop will run until the function returns**(*event*) –

wait_for_value_async(*value*)
Wait for the button to be pressed or released

Parameters

- **value** (*int*) – 1 to wait for press or 0 to wait for release
- **function is an asyncio coroutine**(*This*) –

pynq.board.led module

class pynq.board.led.**LED**(*index*)
Bases: object

This class controls the onboard LEDs.

index
int

The index of the onboard LED, starting from 0.

off()
Turn off a single LED.

Returns

Return type None

on()
Turn on a single LED.

Returns

Return type None

read()
Retrieve the LED state.

Returns Either 0 if the LED is off or 1 if the LED is on.

Return type int

toggle()
Flip the state of a single LED.

If the LED is on, it will be turned off. If the LED is off, it will be turned on.

Returns

Return type None

write(*value*)
Set the LED state according to the input value.

Parameters **value** (*int*) – This parameter can be either 0 (off) or 1 (on).

Raises ValueError – If the value parameter is not 0 or 1.

pynq.board.switch module

class pynq.board.switch.**Switch**(*index*)
Bases: object

This class controls the onboard switches.

index
int

Index of the onboard switches, starting from 0.

read()
Read the current value of the switch.

Returns Either 0 if the switch is off or 1 if the switch is on

Return type int

wait_for_value(*value*)
Wait for the switch to be set to a particular position

Parameters

- **value** (*int*) – 1 for the switch up and 0 for the switch down
- **function wraps the coroutine form so the asyncio** (*This*) –
- **loop will run until the function returns** (*event*) –

wait_for_value_async(*value*)
Wait for the switch to be set to a particular position

Parameters

- **value** (*int*) – 1 for the switch up and 0 for the switch down
- **function is an asyncio coroutine** (*This*) –

pynq.board.rgbled module

class pynq.board.rgbled.**RGBLED**(*index*)
Bases: object

This class controls the onboard RGB LEDs.

index
int

The index of the RGB LED, from 4 (LD4) to 5 (LD5).

_mmio
MMIO

Shared memory map for the RGBLED GPIO controller.

_rgbleds_val
int

Global value of the RGBLED GPIO pins.

off()
Turn off a single RGBLED.

Returns

Return type None

on(*color*)
Turn on a single RGB LED with a color value (see color constants).

Parameters **color** (*int*) – Color of RGB specified by a 3-bit RGB integer value.

Returns

Return type None

read()
Retrieve the RGBLED state.

Returns The color value stored in the RGBLED.

Return type int

write(*color*)
Set the RGBLED state according to the input value.

Parameters **color** (*int*) – Color of RGB specified by a 3-bit RGB integer value.

Returns

Return type None

pynq.iop package

Submodules

pynq.iop.arduino_analog module

class pynq.iop.arduino_analog.**Arduino_Analog**(*if_id*, *gr_pin*)
Bases: object

This class controls the Arduino Analog.

XADC is an internal analog controller in the hardware. This class provides API to do analog reads from IOP.

iop
_IOP

I/O processor instance used by Arduino_Analog class.

mmio
MMIO

Memory-mapped I/O instance to read and write instructions and data.

log_running
int

The state of the log (0: stopped, 1: started).

log_interval_ms
int

Time in milliseconds between samples on the same channel.

gr_pin
list

A list of analog pins getting sampled.

num_channels
int

The number of channels sampled.

get_log()
Return list of logged samples.

Returns List of valid voltage samples (floats) from the ADC sensor.

Return type list

get_log_raw()
Return list of logged raw samples.

Returns List of valid raw samples from the analog device.

Return type list

read()
Read the voltage value from the analog peripheral.

Returns The float values after translation.

Return type list

read_raw()
Read the analog raw value from the analog peripheral.

Returns The raw values from the analog device.

Return type list

reset()
Resets the system monitor for analog devices.

Returns

Return type None

set_log_interval_ms(*log_interval_ms*)
Set the length of the log for the analog peripheral.

This method can set the time interval between two samples, so that users can read out multiple values in a single log.

Parameters **log_interval_ms** (*int*) – The time between two samples in milliseconds, for logging only.

Returns

Return type None

start_log()
Start recording multiple voltage values (float) in a log.

This method will first call set_log_interval_ms() before writing to the MMIO.

Returns

Return type None

start_log_raw()
Start recording raw data in a log.

This method will first call set_log_interval_ms() before writing to the MMIO.

Returns

Return type None

stop_log()
Stop recording the voltage values in the log.

This can be done by calling the stop_log_raw() method.

Returns

Return type None

stop_log_raw()
Stop recording the raw values in the log.

Simply write 0xC to the MMIO to stop the log.

Returns

Return type None

pynq.iop.arduino_io module

class pynq.iop.arduino_io.**Arduino_IO**(*if_id*, *index*, *direction*)
Bases: object

This class controls the Arduino IO pins as inputs or outputs.

Note: The parameter 'direction' determines whether the instance is input/output: 'in' : receiving input from offchip to onchip. 'out' : sending output from onchip to offchip.

Note: The index of the Arduino pins: upper row, from right to left: {0, 1, ..., 13}. (D0 - D13) lower row, from left to right: {14, 15,..., 19}. (A0 - A5)

---

iop
_IOP

The _IOP object returned from the DevMode.

index
int

The index of the Arduino pin, from 0 to 19.

direction
str

Input 'in' or output 'out'.

read()
Receive the value from the offboard Arduino IO device.

Note: Only use this function when direction is 'in'.

Returns The data (0 or 1) on the specified Arduino IO pin.

Return type int

write(*value*)
Send the value to the offboard Arduino IO device.

Note: Only use this function when direction is 'out'.

Parameters **value** (*int*) – The value to be written to the Arduino IO device.

Returns

Return type None

pynq.iop.devmode module

class pynq.iop.devmode.**DevMode**(*if_id*, *switch_config*)
Bases: object

Control an IO processor running the developer mode program.

This class will wait for Python to send commands to IO processor.

if_id
int

The interface ID (1,2,3) corresponding to (PMODA,PMODB,ARDUINO).

iop
_IOP

IO processor instance used by DevMode.

iop_switch_config
list

IO processor switch configuration (8 or 19 integers).

mmio
MMIO

Memory-mapped IO instance to read and write instructions and data.

get_cmd_word(*cmd*, *d_width*, *d_length*)
Build the command word.

Note: The returned command word has the following format: Bit [0] : valid bit. Bit [2:1] : command data width. Bit [3] : command type (read or write). Bit [15:8] : command burst length. Bit [31:16] : unused.

Parameters

- **cmd** (*int*) – Either 1 (read IOP register) or 0 (write IOP register).
- **d_width** (*int*) – Command data width.
- **d_length** (*int*) – Command burst length (currently only supporting d_length 1).

Returns The command word following a specific format.

Return type int

is_cmd_mailbox_idle()
Check whether the IOP command mailbox is idle.

Returns True if IOP command mailbox idle.

Return type bool

load_switch_config(*config=None*)
Load the IO processor's switch configuration.

This method will update switch config.

Parameters **config** (*list*) – A switch configuration list of integers.

Raises TypeError – If the config argument is not of the correct type.

read_cmd(*address*, *d_width=4*, *d_length=1*, *timeout=10*)
Send a read command to the mailbox.

Parameters

- **address** (*int*) – The address tied to IO processor's memory map.
- **d_width** (*int*) – Command data width.
- **d_length** (*int*) – Command burst length (currently only supporting d_length 1).
- **timeout** (*int*) – Time in milliseconds before function exits with warning.

Returns A list of data returned by MMIO read.

Return type list

start()
Start the IO Processor.

The IOP instance will start automatically after instantiation.

This method will: 1. zero out mailbox CMD register; 2. load switch config; 3. set IOP status as "RUNNING".

status()
Returns the status of the IO processor.

Returns The IOP status ("IDLE", "RUNNING", or "STOPPED").

Return type str

stop()
Put the IO Processor into Reset.

This method will set IOP status as "STOPPED".

write_cmd(*address*, *data*, *d_width=4*, *d_length=1*, *timeout=10*)
Send a write command to the mailbox.

Parameters

- **address** (*int*) – The address tied to IO processor's memory map.
- **data** (*int*) – 32-bit value to be written (None for read).
- **d_width** (*int*) – Command data width.
- **d_length** (*int*) – Command burst length (currently only supporting d_length 1).
- **timeout** (*int*) – Time in milliseconds before function exits with warning.

Returns

Return type None

pynq.iop.grove_adc module

class pynq.iop.grove_adc.**Grove_ADC**(*if_id*, *gr_pin*)
Bases: object

This class controls the Grove IIC ADC.

Grove ADC is a 12-bit precision ADC module based on ADC121C021. Hardware version: v1.2.

iop
_IOP

I/O processor instance used by Grove_ADC.

mmio
MMIO

Memory-mapped I/O instance to read and write instructions and data.

log_running
int

The state of the log (0: stopped, 1: started).

log_interval_ms
int

Time in milliseconds between sampled reads of the Grove_ADC sensor.

get_log()
Return list of logged samples.

Returns List of valid voltage samples (floats) from the ADC sensor.

Return type list

get_log_raw()
Return list of logged raw samples.

Returns List of valid raw samples from the ADC sensor.

Return type list

read()
Read the ADC voltage from the Grove ADC peripheral.

Returns The float value after translation.

Return type float

read_raw()
Read the ADC raw value from the Grove ADC peripheral.

Returns The raw value from the sensor.

Return type int

reset()
Resets/initializes the ADC.

Returns

Return type None

set_log_interval_ms(*log_interval_ms*)
Set the length of the log for the Grove_ADC peripheral.

This method can set the time interval between two samples, so that users can read out multiple values in a single log.

Parameters **log_interval_ms** (*int*) – The time between two samples in milliseconds, for logging only.

Returns

Return type None

start_log()
Start recording multiple voltage values (float) in a log.

This method will first call set_log_interval_ms() before writting to the MMIO.

Returns

Return type None

start_log_raw()
Start recording raw data in a log.

This method will first call set_log_interval_ms() before writting to the MMIO.

Returns

Return type None

stop_log()
Stop recording the voltage values in the log.

This can be done by calling the stop_log_raw() method.

Returns

Return type None

stop_log_raw()
Stop recording the raw values in the log.

Simply write 0xC to the MMIO to stop the log.

Returns

Return type None

pynq.iop.grove_buzzer module

class pynq.iop.grove_buzzer.**Grove_Buzzer**(*if_id*, *gr_pin*)
Bases: object

This class controls the Grove Buzzer.

The grove buzzer module has a piezo buzzer as the main component. The piezo can be connected to digital outputs, and will emit a tone when the output is HIGH. Alternatively, it can be connected to an analog pulse-width modulation output to generate various tones and effects. Hardware version: v1.2.

iop
_IOP

I/O processor instance used by Grove Buzzer.

mmio
MMIO

Memory-mapped I/O instance to read and write instructions and data.

log_interval_ms
int

Time in milliseconds between sampled reads of the GROVE_BUZZER sensor.

play_melody()
Play a melody.

Returns

Return type None

play_tone(*tone_period*, *num_cycles*)
Play a single tone with tone_period for num_cycles

Parameters

- **tone_period** (*int*) – The period of the tone in microsecond.
- **num_cycles** (*int*) – The number of cycles for the tone to be played.

Returns

Return type None

pynq.iop.grove_color module

class pynq.iop.grove_color.**Grove_Color**(*if_id*, *gr_pin*)
Bases: object

This class controls the Grove IIC Color sensor.

Grove Color sensor based on the TCS3414CS. Hardware version: v1.3.

iop
_IOP

I/O processor instance used by Grove_Color.

mmio
MMIO

Memory-mapped I/O instance to read and write instructions and data.

log_running
int

The state of the log (0: stopped, 1: started).

read()
Read the color values from the Grove Color peripheral.

The output contains 4 integer values: Red, Green, Blu and Clear. Clear represents the value of the sensor if no color filters are applied.

Returns Tuple of (red, green, blue, clear),

Return type tuple

pynq.iop.grove_dlight module

class pynq.iop.grove_dlight.**Grove_DLight**(*if_id*, *gr_pin*)
Bases: object

This class controls the Grove IIC color sensor.

Grove Color sensor based on the TCS3414CS. Hardware version: v1.3.

iop
_IOP

I/O processor instance used by Grove_Color.

mmio
MMIO

Memory-mapped I/O instance to read and write instructions and data.

log_running
int

The state of the log (0: stopped, 1: started).

read_lux()
Read the computed lux value of the sensor.

Returns The lux value from the sensor

Return type int

read_raw_light()
Read the visible and IR channel values.

Read the values from the grove digital light peripheral.

Returns A tuple containing 2 integer values ch0 (visible) and ch1 (IR).

Return type tuple

pynq.iop.grove_ear_hr module

class pynq.iop.grove_ear_hr.**Grove_EarHR**(*if_id*, *gr_pin*)
Bases: object

This class controls the Grove ear clip heart rate sensor. Sensor model: MED03212P.

iop
_IOP

I/O processor instance used by Grove_FingerHR.

mmio
MMIO

Memory-mapped I/O instance to read and write instructions and data.

read()
Read the heart rate from the sensor.

Returns The heart rate as beats per minute

Return type float

read_raw()
Read the number of heart beats.

Read the number of beats since the sensor initialization; also read the time elapsed in ms between the latest two heart beats.

Returns Number of heart beats and the time elapsed between 2 latest beats.

Return type tuple

pynq.iop.grove_finger_hr module

class pynq.iop.grove_finger_hr.**Grove_FingerHR**(*if_id*, *gr_pin*)
Bases: object

This class controls the Grove finger clip heart rate sensor.

Grove Finger sensor based on the TCS3414CS. Hardware version: v1.3.

iop
_IOP

I/O processor instance used by Grove_FingerHR.

mmio
MMIO

Memory-mapped I/O instance to read and write instructions and data.

log_running
int

The state of the log (0: stopped, 1: started).

get_log()
Return list of logged samples.

Returns List of integers containing the heart rate.

Return type list

read()
Read the heart rate value from the Grove Finger HR peripheral.

Returns A integer representing the heart rate frequency

Return type tuple

start_log(*log_interval_ms=100*)
Start recording multiple heart rate values in a log.

This method will first call set the log interval before writing to the MMIO.

Parameters **log_interval_ms** (*int*) – The time between two samples in milliseconds.

Returns

Return type None

stop_log()
Stop recording the values in the log.

Simply write 0xC to the MMIO to stop the log.

Returns

Return type None

pynq.iop.grove_haptic_motor module

class pynq.iop.grove_haptic_motor.**Grove_Haptic_Motor**(*if_id*, *gr_pin*)
Bases: object

This class controls the Grove Haptic Motor based on the DRV2605L. Hardware version v0.9.

iop
_IOP

I/O processor instance used by Grove_Haptic_Motor.

mmio
MMIO

Memory-mapped I/O instance to read and write instructions and data.

is_playing()
Check if a vibration effect is running on the motor.

Returns True if a vibration effect is playing, false otherwise

Return type bool

play(*effect*)
Play a vibration effect on the Grove Haptic Motor peripheral.

Valid effect identifiers are in the range [1, 127].

Parameters **effect** (*int*) – An integer that specifies the effect.

Returns

Return type None

play_sequence(*sequence*)
Play a sequence of effects possibly separated by pauses.

At most 8 effects or pauses can be specified at a time. Pauses are defined using negative integer values in the range [-1, -127] that correspond to a pause length in the range [10, 1270] ms

Valid effect identifiers are in the range [1, 127]

As an example, in the following sequence example: [4,-20,5] effect 4 is played and after a pause of 200 ms effect 5 is played

Parameters **sequence** (*list*) – At most 8 values specifying effects and pauses.

Returns

Return type None

stop()
Stop an effect or a sequence on the motor peripheral.

Returns

Return type None

pynq.iop.grove_imu module

class pynq.iop.grove_imu.**Grove_IMU**(*if_id*, *gr_pin*)
Bases: object

This class controls the Grove IIC IMU.

Grove IMU 10DOF is a combination of grove IMU 9DOF (MPU9250) and grove barometer sensor (BMP180). MPU-9250 is a 9-axis motion tracking device that combines a 3-axis gyroscope, 3-axis accelerometer, 3-axis magnetometer and a Digital Motion Processor (DMP). BMP180 is a high precision, low power digital pressure sensor. Hardware version: v1.1.

iop
_IOP

I/O processor instance used by Grove_IMU.

mmio
MMIO

Memory-mapped I/O instance to read and write instructions and data.

get_accl()
Get the data from the accelerometer.

Returns A list of the acceleration data along X-axis, Y-axis, and Z-axis.

Return type list

get_altitude()
Get the current altitude.

Returns The altitude value.

Return type float

get_atm()
Get the current pressure in relative atmosphere.

Returns The related atmosphere.

Return type float

get_compass()
Get the data from the magnetometer.

Returns A list of the compass data along X-axis, Y-axis, and Z-axis.

Return type list

get_gyro()
Get the data from the gyroscope.

Returns A list of the gyro data along X-axis, Y-axis, and Z-axis.

Return type list

get_heading()
Get the value of the heading.

Returns The angle deviated from the X-axis, toward the positive Y-axis.

Return type float

get_pressure()
Get the current pressure in Pa.

Returns The pressure value.

Return type float

get_temperature()
Get the current temperature in degree C.

Returns The temperature value.

Return type float

get_tilt_heading()
Get the value of the tilt heading.

Returns The tilt heading value.

Return type float

reset()
Reset all the sensors on the grove IMU.

Returns

Return type None

pynq.iop.grove_ledbar module

class pynq.iop.grove_ledbar.**Grove_LEDbar**(*if_id*, *gr_pin*)
Bases: object

This class controls the Grove LED BAR.

Grove LED Bar is comprised of a 10 segment LED gauge bar and an MY9221 LED controlling chip. Model: LED05031P. Hardware version: v2.0.

iop
_IOP

I/O processor instance used by Grove_LEDbar.

mmio
MMIO

Memory-mapped I/O instance to read and write instructions and data.

read()
Reads the current status of LEDbar.

Reads the current status of LED bar and returns 10-bit binary string. Each bit position corresponds to a LED position in the LEDbar, and bit value corresponds to the LED state.

Red LED corresponds to the LSB, while green LED corresponds to the MSB.

Returns String of 10 binary bits.

Return type str

reset()
Resets the LEDbar.

Clears the LED bar, sets all LEDs to OFF state.

Returns

Return type None

write_binary(*data_in*)
Set individual LEDs in the LEDbar based on 10 bit binary input.

Each bit in the 10-bit *data_in* points to a LED position on the LEDbar. Red LED corresponds to the LSB, while green LED corresponds to the MSB.

Parameters **data_in** (*int*) – 10 LSBs of this parameter control the LEDbar.

Returns

Return type None

write_brightness(*data_in, brightness=[170, 170, 170, 170, 170, 170, 170, 170, 170, 170]*)
Set individual LEDs with 3 level brightness control.

Each bit in the 10-bit *data_in* points to a LED position on the LEDbar. Red LED corresponds to the LSB, while green LED corresponds to the MSB.

Brightness of each LED is controlled by the brightness parameter. There are 3 perceivable levels of brightness: 0xFF : HIGH 0xAA : MED 0x01 : LOW

Parameters

- **data_in** (*int*) – 10 LSBs of this parameter control the LEDbar.
- **brightness** (*list*) – Each List element controls a single LED.

Returns

Return type None

write_level(*level*, *bright_level*, *green_to_red*)
Set the level to which the leds are to be lit in levels 1 - 10.

Level can be set in both directions. *set_level* operates by setting all LEDs to the same brightness level.

There are 4 preset brightness levels: bright_level = 0: off bright_level = 1: low bright_level = 2: medium bright_level = 3: maximum

green_to_red indicates the direction, either from red to green when it is 0, or green to red when it is 1.

Parameters

- **level** (*int*) – 10 levels exist, where 1 is minimum and 10 is maximum.
- **bright_level** (*int*) – Controls brightness of all LEDs in the LEDbar, from 0 to 3.
- **green_to_red** (*int*) – Sets the direction of the sequence.

Returns

Return type None

pynq.iop.grove_light module

class pynq.iop.grove_light.**Grove_Light**(*if_id*, *gr_pin*)
Bases: *pynq.iop.grove_adc.Grove_ADC*

This class controls the grove light sensor.

This class inherits from the Grove_ADC class. To use this module, grove ADC has to be used as a bridge. The light sensor incorporates a Light Dependent Resistor (LDR) GL5528. Hardware version: v1.1.

iop
_IOP

I/O processor instance used by Grove ADC.

mmio
MMIO

Memory-mapped I/O instance to read and write instructions and data.

log_running
int

The state of the log (0: stopped, 1: started).

log_interval_ms
int

Time in milliseconds between sampled reads of the Grove ADC sensor.

get_log()
Return list of logged light sensor resistances.

Returns List of valid light sensor resistances.

Return type list

read()
Read the light sensor resistance in from the light sensor.

This method overrides the definition in Grove_ADC.

Returns The light reading in terms of the sensor resistance.

Return type float

start_log()
Start recording the light sensor resistance in a log.

This method will call the start_log_raw() in the parent class.

Returns

Return type None

stop_log()
Stop recording light values in a log.

This method will call the stop_log_raw() in the parent class.

Returns

Return type None

pynq.iop.grove_oled module

class pynq.iop.grove_oled.**Grove_OLED**(*if_id*, *gr_pin*)
Bases: object

This class controls the Grove IIC OLED.

Grove LED 128×64 Display module is an OLED monochrome 128×64 matrix display module. Model: OLE35046P. Hardware version: v1.1.

iop
_IOP

I/O processor instance used by Grove_OLED.

mmio
MMIO

Memory-mapped I/O instance to read and write instructions and data.

clear()
Clear the OLED screen.

This is done by writing empty strings into the OLED in Microblaze.

Returns

Return type None

set_contrast(*brightness*)
Set the contrast level for the OLED display.

The contrast level is in [0, 255].

Parameters **brightness** (*int*) – The brightness of the display.

Returns

Return type None

set_horizontal_mode()
Set the display mode to horizontal.

Returns

Return type None

set_inverse_mode()
Set the display mode to inverse.

Returns

Return type None

set_normal_mode()
Set the display mode to normal.

Returns

Return type None

set_page_mode()
Set the display mode to paged.

Returns

Return type None

set_position(*row*, *column*)
Set the position of the display.

The position is indicated by (row, column).

Parameters

- **row** (*int*) – The row number to start the display.
- **column** (*int*) – The column number to start the display.

Returns

Return type None

write(*text*)
Write a new text string on the OLED.

Clear the screen first to correctly show the new text.

Parameters **text** (*str*) – The text string to be displayed on the OLED screen.

Returns

Return type None

pynq.iop.grove_pir module

class pynq.iop.grove_pir.**Grove_PIR**(*if_id*, *gr_pin*)
Bases: object

This class controls the PIR motion sensor.

The grove PIR motion sensor is attached to a Pmod or an Arduino interface. Hardware version: v1.2.

pir_iop
object

The Pmod IO or Arduino IO object.

read()
Receive the value from the PIR sensor.

Returns 0 when there is no motion, and returns 1 otherwise.

Returns The data (0 or 1) read from the PIR sensor.

Return type int

pynq.iop.grove_th02 module

class pynq.iop.grove_th02.**Grove_TH02**(*if_id*, *gr_pin*)
Bases: object

This class controls the Grove I2C Temperature and Humidity sensor.

Temperature & humidity sensor (high-accuracy & mini). Hardware version: v1.0.

iop
_IOP

I/O processor instance used by Grove_TH02.

mmio
MMIO

Memory-mapped I/O instance to read and write instructions and data.

log_running
int

The state of the log (0: stopped, 1: started).

get_log()
Return list of logged samples.

Returns List of tuples containing (temperature, humidity)

Return type list

read()
Read the temperature and humidity values from the TH02 peripheral.

Returns tuple containing (temperature, humidity)

Return type tuple

start_log(*log_interval_ms=100*)
Start recording multiple heart rate values in a log.

This method will first call set the log interval before writting to the MMIO.

Parameters **log_interval_ms** (*int*) – The time between two samples in milliseconds.

Returns

Return type None

stop_log()
Stop recording the values in the log.

Simply write 0xC to the MMIO to stop the log.

Returns

Return type None

pynq.iop.grove_tmp module

class pynq.iop.grove_tmp.**Grove_TMP**(*if_id*, *gr_pin*, *version='v1.2'*)
Bases: *pynq.iop.grove_adc.Grove_ADC*

This class controls the grove temperature sensor.

This class inherits from the Grove_ADC class. To use this module, grove ADC has to be used as a bridge. The temperature sensor uses a thermistor to detect the ambient temperature. Hardware version: v1.2.

iop
_IOP

I/O processor instance used by Grove ADC.

mmio
MMIO

Memory-mapped I/O instance to read and write instructions and data.

log_running
int

The state of the log (0: stopped, 1: started).

log_interval_ms
int

Time in milliseconds between sampled reads of the Grove ADC sensor.

bValue
int

The thermistor constant.

get_log()
Return list of logged temperature samples.

Returns List of valid temperature readings from the temperature sensor.

Return type list

read()
Read temperature values in Celsius from temperature sensor.

This method overrides the definition in Grove_ADC.

Returns The temperature reading in Celsius.

Return type float

start_log()
Start recording temperature in a log.

This method will call the start_log_raw() in the parent class.

Parameters **None** –

Returns

Return type None

stop_log()
Stop recording temperature in a log.

This method will call the stop_log_raw() in the parent class.

Returns

Return type None

pynq.iop.iop module

pynq.iop.iop.**request_iop**(*iop_id*, *mb_program*)
This is the interface to request an I/O Processor.

It looks for active instances on the same IOP ID, and prevents users from instantiating different types of IOPs on the same interface. Users are notified with an exception if the selected interface is already hooked to another type of IOP, to prevent unwanted behavior.

Two cases: 1. No previous IOP in the system with the same ID, or users want to request another instance with the same program. Do not raises an exception. 2. There is A previous IOP in the system with the same ID. Users want to request another instance with a different program. Raises an exception.

Note: When an IOP is already in the system with the same IOP ID, users are in danger of losing the old instances associated with this IOP.

For bitstream *base.bit*, the IOP IDs are {1, 2, 3} <=> {PMODA, PMODB, arduino interface}. For different bitstreams, this mapping can be different.

Parameters

- **iop_id** (*int*) – IOP ID (1, 2, 3) corresponding to (PMODA, PMODB, ARDUINO).
- **mb_program** (*str*) – Program to be loaded on the IOP.

Returns An _IOP object with the updated Microblaze program.

Return type _IOP

Raises

- ValueError – When the IOP name or the GPIO name cannot be found in the PL.
- LookupError – When another IOP is in the system with the same IOP ID.

pynq.iop.iop_const module

pynq.iop.pmod_adc module

class pynq.iop.pmod_adc.**Pmod_ADC**(*if_id*)
Bases: object

This class controls an Analog to Digital Converter Pmod.

The Pmod AD2 (PB 200-217) is an analog-to-digital converter powered by AD7991. Users may configure up to 4 conversion channels at 12 bits of resolution.

iop
_IOP

I/O processor instance used by the ADC

mmio
MMIO

Memory-mapped I/O instance to read and write instructions and data.

log_running
int

The state of the log (0: stopped, 1: started).

get_log()
Get the log of voltage values.

First stop the log before getting the log.

Returns List of voltage samples from the ADC.

Return type list

get_log_raw()
Get the log of raw values.

First stop the log before getting the log.

Returns List of raw samples from the ADC.

Return type list

read(*ch1=1*, *ch2=0*, *ch3=0*)
Get the voltage from the Pmod ADC.

When ch1, ch2, and ch3 values are 1 then the corresponding channel is included.

For each channel selected, this method reads and returns one sample.

Note: The 4th channel is not available due to the jumper setting on ADC.

Note: This method reads the voltage values from ADC.

Parameters

- **ch1** (*int*) – 1 means include channel 1, 0 means do not include.
- **ch2** (*int*) – 1 means include channel 2, 0 means do not include.
- **ch3** (*int*) – 1 means include channel 3, 0 means do not include.

Returns The voltage values read from the 3 channels of the Pmod ADC.

Return type list

read_raw(*ch1=1*, *ch2=0*, *ch3=0*)
Get the raw value from the Pmod ADC.

When ch1, ch2, and ch3 values are 1 then the corresponding channel is included.

For each channel selected, this method reads and returns one sample.

Note: The 4th channel is not available due to the jumper (JP1) setting on ADC.

Note: This method reads the raw value from ADC.

Parameters

- **ch1** (*int*) – 1 means include channel 1, 0 means do not include.
- **ch2** (*int*) – 1 means include channel 2, 0 means do not include.
- **ch3** (*int*) – 1 means include channel 3, 0 means do not include.

Returns The raw values read from the 3 channels of the Pmod ADC.

Return type list

reset()
Reset the Pmod ADC.

Returns

Return type None

start_log(*ch1=1*, *ch2=0*, *ch3=0*, *log_interval_us=100*)
Start the log of voltage values with the interval specified.

This parameter *log_interval_us* can set the time interval between two samples, so that users can read out multiple values in a single log.

Parameters

- **ch1** (*int*) – 1 means include channel 1, 0 means do not include.
- **ch2** (*int*) – 1 means include channel 2, 0 means do not include.
- **ch3** (*int*) – 1 means include channel 3, 0 means do not include.
- **log_interval_us** (*int*) – The length of the log in milliseconds, for debug only.

Returns

Return type None

start_log_raw(*ch1=1*, *ch2=0*, *ch3=0*, *log_interval_us=100*)
Start the log of raw values with the interval specified.

This parameter *log_interval_us* can set the time interval between two samples, so that users can read out multiple values in a single log.

Parameters

- **ch1** (*int*) – 1 means include channel 1, 0 means do not include.
- **ch2** (*int*) – 1 means include channel 2, 0 means do not include.
- **ch3** (*int*) – 1 means include channel 3, 0 means do not include.
- **log_interval_us** (*int*) – The length of the log in milliseconds, for debug only.

Returns

Return type None

stop_log()
Stop the log of voltage values.

This is done by sending the reset command to IOP. There is no need to wait for the IOP.

Returns

Return type None

stop_log_raw()
Stop the log of raw values.

This is done by sending the reset command to IOP. There is no need to wait for the IOP.

Returns

Return type None

pynq.iop.pmod_als module

class pynq.iop.pmod_als.**Pmod_ALS**(*if_id*)
Bases: object

This class controls a light sensor Pmod.

The Digilent Pmod ALS demonstrates light-to-digital sensing through a single ambient light sensor. This is based on an ADC081S021 analog-to-digital converter and a TEMT6000X01 ambient light sensor.

iop
_IOP

I/O processor instance used by ALS

mmio
MMIO

Memory-mapped I/O instance to read and write instructions and data.

log_interval_ms
int

Time in milliseconds between sampled reads of the ALS sensor

get_log()
Return list of logged samples.

Returns

Return type List of valid samples from the ALS sensor [0-255]

read()
Read current light value measured by the ALS Pmod.

Returns The current sensor value.

Return type int

set_log_interval_ms(*log_interval_ms*)
Set the length of the log in the ALS Pmod.

This method can set the length of the log, so that users can read out multiple values in a single log.

Parameters **log_interval_ms** (*int*) – The length of the log in milliseconds, for debug only.

Returns

Return type None

start_log()
Start recording multiple values in a log.

This method will first call set_log_interval_ms() before writting to the MMIO.

Returns

Return type None

stop_log()
Stop recording multiple values in a log.

Simply write to the MMIO to stop the log.

Returns

Return type None

pynq.iop.pmod_cable module

class pynq.iop.pmod_cable.**Pmod_Cable**(*if_id*, *index*, *direction*, *cable*)
Bases: *pynq.iop.pmod_io.Pmod_IO*

This class can be used for a cable connecting Pmod interfaces.

This class inherits from the Pmod IO class.

Note: When 2 Pmods are connected using a cable, the parameter 'cable' decides whether the cable is a 'loopback' or 'straight' cable. The default is a straight cable (no internal wire twisting). For pin mapping, please check the Pmod IO class.

---

iop
_IOP

The _IOP object returned from the DevMode.

index
int

The index of the Pmod pin, from 0 to 7.

direction
str

Input 'in' or output 'out'.

cable
str

Either 'straight' or 'loopback'.

read()
Receive the value from the Pmod cable.

This class overrides the read() method in the Pmod IO class.

Note: Only use this function when direction = 'in'.

When two Pmods are connected on the same board, for any received raw value, a "straignt" cable flips the upper 4 pins and the lower 4 pins: A Pmod interface <=> Another Pmod interface {vdd,gnd,3,2,1,0} <=> {vdd,gnd,7,6,5,4} {vdd,gnd,7,6,5,4} <=> {vdd,gnd,3,2,1,0}

A "loop-back" cable satisfies the following mapping between two Pmods: A Pmod interface <=> Another Pmod interface {vdd,gnd,3,2,1,0} <=> {vdd,gnd,3,2,1,0} {vdd,gnd,7,6,5,4} <=> {vdd,gnd,7,6,5,4}

Returns The data (0 or 1) on the specified Pmod IO pin.

Return type int

set_cable(*cable*)
Set the type for the cable.

Note: The default cable type is 'straight'. Only straight cable or loop-back cable can be recognized.

Parameters **cable** (*str*) – Either 'straight' or 'loopback'.

Returns

Return type None

pynq.iop.pmod_dac module

class pynq.iop.pmod_dac.**Pmod_DAC**(*if_id*, *value=None*)
Bases: object

This class controls a Digital to Analog Converter Pmod.

The Pmod DA4 (PB 200-245) is an 8 channel 12-bit digital-to-analog converter run via AD5628.

iop
_IOP

I/O processor instance used by the DAC

mmio
MMIO

Memory-mapped I/O instance to read and write instructions and data.

write(*value*)
Write a floating point number onto the DAC Pmod.

Note: User is not allowed to use a number outside of the range [0.00, 2.00] as the input value.

Parameters **value** (*float*) – The value to be written to the DAC Pmod

Returns

Return type None

pynq.iop.pmod_dpot module

class pynq.iop.pmod_dpot.**Pmod_DPOT**(*if_id*)
Bases: object

This class controls a digital potentiometer Pmod.

The Pmod DPOT (PB 200-239) is a digital potentiometer powered by the AD5160. Users may set a desired resistance between 60 ~ 10k ohms.

iop
_IOP

I/O processor instance used by DPOT

mmio
MMIO

Memory-mapped I/O instance to read and write instructions and data.

write(*val*, *step=0*, *log_ms=0*)
Write the value into the DPOT.

This method will write the parameters "value", "step", and "log_ms" all together into the DPOT Pmod. The parameter "log_ms" is only used for debug; users can ignore this parameter.

Parameters

- **val** (*int*) – The initial value to start, in [0, 255].
- **step** (*int*) – The number of steps when ramping up to the final value.
- **log_ms** (*int*) – The length of the log in milliseconds, for debug only.

Returns

Return type None

pynq.iop.pmod_iic module

class pynq.iop.pmod_iic.**Pmod_IIC**(*if_id*, *scl_pin*, *sda_pin*, *iic_addr*)
Bases: object

This class controls the Pmod IIC pins.

Note: The index of the Pmod pins: upper row, from left to right: {vdd,gnd,3,2,1,0}. lower row, from left to right: {vdd,gnd,7,6,5,4}.

iop
_IOP

The _IOP object returned from the DevMode.

scl_pin
int

The SCL pin number.

sda_pin
int

The SDA pin number.

iic_addr
int

The IIC device address.

sr_addr
int

The IIC device SR address (base address + 0x104).

dtr_addr
int

The IIC device DTR address (base address + 0x108).

cr_addr
int

The IIC device CR address (base address + 0x100).

rfd_addr
int

The IIC device RFD address (base address + 0x120).

drr_addr
int

The IIC device DRR address (base address + 0x10C).

receive(*num_bytes*)
This method receives IIC bytes from the device.

Parameters **num_bytes** (*int*) – Number of bytes to be received from the device.

Returns **iic_bytes** – A list of 8-bit bytes received from the driver.

Return type list

Raises `RuntimeError` – Timeout when waiting for the RX FIFO to fill.

send(*iic_bytes*)
This method sends the command or data to the driver.

Parameters **iic_bytes** (*list*) – A list of 8-bit bytes to be sent to the driver.

Returns

Return type None

Raises `RuntimeError` – Timeout when waiting for the FIFO to be empty.

pynq.iop.pmod_io module

class pynq.iop.pmod_io.**Pmod_IO**(*if_id*, *index*, *direction*)
Bases: object

This class controls the Pmod IO pins as inputs or outputs.

Note: The parameter 'direction' determines whether the instance is input/output: 'in' : receiving input from offchip to onchip. 'out' : sending output from onchip to offchip. The index of the Pmod pins: upper row, from left to right: {vdd,gnd,3,2,1,0}. lower row, from left to right: {vdd,gnd,7,6,5,4}.

---

iop
_IOP

The _IOP object returned from the DevMode.

index
int

The index of the Pmod pin, from 0 to 7.

direction
str

Input 'in' or output 'out'.

read()
Receive the value from the offboard Pmod IO device.

Note: Only use this function when direction is 'in'.

Returns The data (0 or 1) on the specified Pmod IO pin.

Return type int

write(*value*)
Send the value to the offboard Pmod IO device.

Note: Only use this function when direction is 'out'.

Parameters **value** (*int*) – The value to be written to the Pmod IO device.

Returns

Return type None

pynq.iop.pmod_led8 module

class pynq.iop.pmod_led8.**Pmod_LED8**(*if_id*, *index*)
Bases: object

This class controls a single LED on the LED8 Pmod.

The Pmod LED8 (PB 200-163) has eight high-brightness LEDs. Each LED can be individually illuminated from a logic high signal.

iop
_IOP

I/O processor instance used by LED8.

index
int

Index of the pin on LED8, from 0 to 7.

off()
Turn off a single LED.

Returns

Return type None

on()
Turn on a single LED.

Returns

Return type None

read()
Retrieve the LED state.

Returns The data (0 or 1) read out from the selected pin.

Return type int

toggle()
Flip the bit of a single LED.

Note: The LED will be turned off if it is on. Similarly, it will be turned on if it is off.

Returns

Return type None

write(*value*)
Set the LED state according to the input value

Note: This method does not take into account the current LED state.

Parameters **value** (*int*) – Turn on the LED if value is 1; turn it off if value is 0.

Returns

Return type None

pynq.iop.pmod_oled module

class pynq.iop.pmod_oled.**Pmod_OLED**(*if_id*, *text=None*)
Bases: object

This class controls an OLED Pmod.

The Pmod OLED (PB 200-222) is 128x32 pixel monochrome organic LED (OLED) panel powered by the Solomon Systech SSD1306.

iop
_IOP

I/O processor instance used by the OLED

mmio
MMIO

Memory-mapped I/O instance to read and write instructions and data.

clear()
Clear the OLED screen.

This is done by sending the clear command to the IOP.

Returns

Return type None

draw_line(*x1*, *y1*, *x2*, *y2*)
Draw a straight line on the OLED.

Parameters

- **x1** (*int*) – The x-position of the starting point.
- **y1** (*int*) – The y-position of the starting point.
- **x2** (*int*) – The x-position of the ending point.
- **y2** (*int*) – The y-position of the ending point.

Returns

Return type None

draw_rect(*x1*, *y1*, *x2*, *y2*)
Draw a rectangle on the OLED.

Parameters

- **x1** (*int*) – The x-position of the starting point.
- **y1** (*int*) – The y-position of the starting point.
- **x2** (*int*) – The x-position of the ending point.
- **y2** (*int*) – The y-position of the ending point.

Returns

Return type None

write(*text*, *x=0*, *y=0*)
Write a new text string on the OLED.

Parameters

- **text** (*str*) – The text string to be displayed on the OLED screen.
- **x** (*int*) – The x-position of the display.
- **y** (*int*) – The y-position of the display.

Returns

Return type None

pynq.iop.pmod_pwm module

class pynq.iop.pmod_pwm.**Pmod_PWM**(*if_id*, *index*)
Bases: object

This class uses the PWM of the IOP.

iop
_IOP

I/O processor instance used by Pmod_PWM.

mmio
MMIO

Memory-mapped I/O instance to read and write instructions and data.

generate(*period*, *duty_cycle*)
Generate pwm signal with desired period and percent duty cycle.

Parameters

- **period** (*int*) – The period of the tone (us), between 1 and 65536.
- **duty_cycle** (*int*) – The duty cycle in percentage.

Returns

Return type None

stop()
Stops PWM generation.

Returns

Return type None

pynq.iop.pmod_tc1 module

class pynq.iop.pmod_tc1.**Pmod_TC1**(*if_id*)
Bases: object

This class controls a thermocouple Pmod.

The Digilent PmodTC1 is a cold-junction thermocouple-to-digital converter module designed for a classic K-Type thermocouple wire. With Maxim Integrated's MAX31855, this module reports the measured temperature in 14-bits with 0.25 degC resolution.

iop
_IOP

I/O processor instance used by TC1

mmio
MMIO

Memory-mapped I/O instance to read and write instructions and data.

log_interval_ms
int

Time in milliseconds between sampled reads of the TC1 sensor

get_log()
Return list of logged samples.

Returns

Return type List of valid samples from the TC1 sensor

read()
Read full 32-bit register of TC1 Pmod.

Returns The current register contents.

Return type int

reg_to_alarms(*reg_val*)
Extracts Alarm flags from 32-bit register value.

Parameters **reg_val** (*int*) – 32-bit TC1 register value

Returns The alarm flags from the TC1. bit 0 = 1 if thermocouple connection is open-circuit; bit 1 = 1 if thermocouple connection is shorted to generated; bit 2 = 1 if thermocouple connection is shorted to VCC; bit 16 = 1 if any if bits 0-2 are 1.

Return type u32

reg_to_ref(*reg_val*)
Extracts Ref Junction temperature from 32-bit register value.

Parameters **reg_val** (*int*) – 32-bit TC1 register value

Returns The reference junction temperature in degC.

Return type float

reg_to_tc(*reg_val*)
Extracts Thermocouple temperature from 32-bit register value.

Parameters **reg_val** (*int*) – 32-bit TC1 register value

Returns The thermocouple temperature in degC.

Return type float

set_log_interval_ms(*log_interval_ms*)
Set the length of the log in the TC1 Pmod.

This method can set the length of the log, so that users can read out multiple values in a single log.

Parameters **log_interval_ms** (*int*) – The length of the log in milliseconds, for debug only.

Returns

Return type None

start_log()
Start recording multiple values in a log.

This method will first call set_log_interval_ms() before writting to the MMIO.

Returns

Return type None

stop_log()
Stop recording multiple values in a log.

Simply write to the MMIO to stop the log.

Returns

Return type None

pynq.iop.pmod_timer module

class pynq.iop.pmod_timer.**Pmod_Timer**(*if_id*, *index*)
Bases: object

This class uses the timer's capture and generation capabilities.

iop
_IOP

I/O processor instance used by Pmod_Timer.

mmio
MMIO

Memory-mapped I/O instance to read and write instructions and data.

clk
int

The clock period of the IOP in ns.

event_count(*period*)
Count the number of rising edges detected in (period) clocks.

Parameters **period** (*int*) – The period of the generated signals.

Returns The number of events detected.

Return type int

event_detected(*period*)
Detect a rising edge or high-level in (period) clocks.

Parameters **period** (*int*) – The period of the generated signals.

Returns 1 if any event is detected, and 0 if no event is detected.

Return type int

generate_pulse(*period*, *times=0*)
Generate pulses every (period) clocks for a number of times.

The default is to generate pulses every (period) IOP clocks forever until stopped. The pulse width is equal to the IOP clock period.

Parameters

- **period** (*int*) – The period of the generated signals.
- **times** (*int*) – The number of times for which the pulses are generated.

Returns

Return type None

get_period_ns()
Measure the period between two successive rising edges.

Returns Measured period in ns.

Return type int

stop()
This method stops the timer.

Returns

Return type None

pynq.iop.pmod_tmp2 module

class pynq.iop.pmod_tmp2.**Pmod_TMP2**(*if_id*)
Bases: object

This class controls a temperature sensor Pmod.

The Pmod TMP2 (PB 200-221) is an ambient temperature sensor powered by ADT7420.

iop
_IOP

I/O processor instance used by TMP2.

mmio
MMIO

Memory-mapped I/O instance to read and write instructions and data.

log_interval_ms
int

Time in milliseconds between sampled reads of the TMP2 sensor.

get_log()
Return list of logged samples.

Returns

Return type List of valid samples from the temperature sensor in Celsius.

read()
Read current temperature value measured by the Pmod TMP2.

Returns The current sensor value.

Return type float

set_log_interval_ms(*log_interval_ms*)
Set the sampling interval for the Pmod TMP2.

Parameters **log_interval_ms** (*int*) – Time in milliseconds between sampled reads of the TMP2 sensor

Returns

Return type None

start_log()
Start recording multiple values in a log.

This method will first call set_log_interval_ms() before writting to the MMIO.

Returns

Return type None

stop_log()
Stop recording multiple values in a log.

Simply write to the MMIO to stop the log.

Returns

Return type None

pynq.drivers package

Submodules

pynq.drivers.audio module

class pynq.drivers.audio.**Audio**(*ip='SEG_d_axi_pdm_1_S_AXI_reg'*, *rst='audio_path_sel'*)
Bases: object

Class to interact with audio controller.

Each audio sample is a 32-bit integer. The audio controller supports only mono mode, and uses pulse density modulation (PDM).

mmio
MMIO

The MMIO object associated with the audio controller.

gpio
GPIO

The GPIO object associated with the audio controller.

buffer
numpy.ndarray

The numpy array to store the audio.

sample_rate
int

Sample rate of the current buffer content.

sample_len
int

Sample length of the current buffer content.

bypass_start()
Stream audio controller input directly to output.

Returns

Return type None

bypass_stop()
Stop streaming input to output directly.

Returns

Return type None

static **info** (*file*)
Prints information about pdm files.

The information includes name, channels, samples, frames, etc.

Note: The file will be searched in the specified path, or in the working directory in case the path does not exist.

Parameters **file** (*string*) – File name, with a default extension of *pdm*.

Returns

Return type None

load (*file*)
Loads file into internal audio buffer.

The recorded file is of format **.pdm*.

Note: The file will be searched in the specified path, or in the working directory in case the path does not exist.

Parameters **file** (*string*) – File name, with a default extension of *pdm*.

Returns

Return type None

play ()
Play audio buffer via audio jack.

Returns

Return type None

record (*seconds*)
Record data from audio controller to audio buffer.

The sample rate per word is 192000Hz.

Parameters **seconds** (*float*) – The number of seconds to be recorded.

Returns

Return type None

save (*file*)
Save audio buffer content to a file.

The recorded file is of format **.pdm*.

Note: The saved file will be put into the specified path, or in the working directory in case the path does not exist.

Parameters **file** (*string*) – File name, with a default extension of *pdm*.

Returns

Return type None

pynq.drivers.dma module

class pynq.drivers.dma.**DMA**(*address*, *direction=1*, *attr_dict=None*)
Bases: object

Python class which controls DMA.

This is a generic DMA class that can be used to access main memory.

The DMA direction can be:

(0)'DMA_TO_DEV' : DMA sends data to PL.

(1)'DMA_FROM_DEV' : DMA receives data from PL.

(3)'DMA_BIDIRECTIONAL' : DMA can send/receive data from PL.

buf
cffi.FFI.CData

A pointer to physically contiguous buffer.

bufLength
int

Length of internal buffer in bytes.

phyAddress
int

Physical address of the DMA device.

DMAengine
*cdata 'XAxiDma *'*

DMA engine instance defined in C. Not to be directly modified.

DMAinstance
*cdata 'XAxiDma_Config *'*

DMA configuration instance struct. Not to be directly modified.

direction
int

The direction indicating whether DMA sends/receives data from PL.

Configuration
dict

Current DMAinstance configuration values.

__del__()
Destructor for DMA object.

Frees the internal buffer and Resets the DMA.

Parameters **None** –

Returns

Return type None

configure(*attr_dict=None*)
Reconfigure and Reinitialize the DMA IP.

Uses a user provided dict to reinitialize the DMA. This method also frees the internal buffer associated with current object.

The keys in *attr_dict* should exactly match the ones used in default config. All the keys are not required. The default configuration is defined in dma.DefaultConfig dict. Users can reinitialize the DMA with new configuratiuon after creating the object.

Parameters **attr_dict** (*dict*) – A dictionary specifying DMA configuration values.

Returns

Return type None

create_buf(*num_bytes*, *cacheable=0*)
Allocate physically contiguous memory buffer.

Allocates/Reallocates buffer needed for DMA operations.

Possible values for parameter *cacheable* are:

1: the memory buffer is cacheable.

0: the memory buffer is non-cacheable.

Note: This buffer is allocated inside the kernel space using xlnk driver. The maximum allocatable memory is defined at kernel build time using the CMA memory parameters. For Pynq-Z1 kernel, it is specified as 128MB.

Parameters

- **num_bytes** (*int*) – Length of the allocated array in bytes.
- **cacheable** (*int*) – Indicating whether or not the memory buffer is cacheable

Returns

Return type None

free_buf()
Free the memory buffer associated with this object.

Use this to free a previously allocated memory buffer. This is specially useful for reallocations.

Parameters **None** –

Returns

Return type None

get_buf(*width=32*)
Get a CFFI pointer to object's internal buffer.

This can be accessed like a regular array in python. The width can be either 32 or 64.

Parameters **width** (*int*) – The data width in the buffer.

Returns An CFFI object which can be accessed similar to arrays in C.

Return type cffi.FFI.CData

transfer(*num_bytes*, *direction=1*)
Transfer data using DMA (Non-blocking).

Used to initiate transfer of data between a physically contiguous buffer and PL. The buffer should be allocated using *create_buf* before this call.

The *num_bytes* should be less than buffer size and *DMA_TRANSFER_LIMIT_BYTES*.

Possible values for *direction* are:

(0)'DMA_TO_DEV' : DMA sends data to PL.

(1)'DMA_FROM_DEV' : DMA receives data from PL.

Parameters

- **num_bytes** (*int*) – Number of bytes to transfer.
- **direction** (*int*) – Direction in which DMA transfers data.

Returns

Return type None

wait(*wait_timeout=10*)
Block till DMA is busy or a timeout occurs.

Default value of timeout is 10 seconds.

Parameters **wait_timeout** (*int*) – Time to wait in seconds before timing out wait operation.

Returns

Return type None

class pynq.drivers.dma.**timeout**(*seconds=1*, *error_message='Timeout'*)
Bases: object

Internal timeout functions.

This class is only used internally.

handle_timeout(*signum*, *frame*)

pynq.drivers.trace_buffer module

class pynq.drivers.trace_buffer.**Trace_Buffer**(*if_id*, *pins*, *protocol*, *probes=None*, *trace=None*, *rate=500000*)
Bases: object

Class for the trace buffer, leveraging the sigrok libraries.

This trace buffer class gets the traces from DMA and processes it using the sigrok commands.

For PMODA and PMODB, pin numbers 0-7 correspond to the pins on the Pmod interface. Although PMODA and PMODB are sharing the same trace buffer, only one Pmod can be traced at a specific time.

For ARDUINO, pin numbers 0-5 correspond to A0-A5; pin numbers 6-7 correspond to D0-D1; pin numbers 8-19 correspond to D2-D13; pin numbers 20-21 correspond to SDA and SCL.

if_id
int

The interface ID (PMODA, PMODB, ARDUINO).

pins
list

Array of pin numbers, 0-7 for PMODA or PMODB and 0-21 for ARDUINO.

protocol
str

The protocol the sigrok decoder are using, for example, I2C.

trace_csv
str

Absolute path of the **.csv* trace that can be opened by text editor.

trace_sr
str

Absolute path of the **.sr* trace file that can be unzipped.

trace_pd
str

Absolute path of the **.pd* decoded file by sigrok decoder.

probes
list

The list of probes used for the trace, e.g., ['SCL','SDA'] for I2C.

dma
DMA

The DMA object associated with the trace buffer.

ctrl
MMIO

The MMIO class used to control the DMA.

rate
int

The sample rate of the traces, at most 100M samples per second.

samples
ndarray

The np array storing the 64-bit samples.

ffi
cffi.api.FFI

The FFI API to the underlying C structure

__del__()
Destructor for trace buffer object.

Returns

Return type None

csv2sr()
Translate the **.csv* file to **.sr* file.

The translated **.sr* files can be directly used in PulseView to show the waveform.

Note: This method also modifies the input **.csv* file (the comment header, usually 3 lines, will be removed).

Returns

Return type None

decode(*decoded_file*, *options=''*)

Decode and record the trace based on the protocol specified.

The *decoded_file* contains the name of the output file.

The *option* specifies additional options to be passed to sigrok-cli. For example, users can use option=':wordsize=9:cpol=1:cpha=0' to add these options for the SPI decoder.

The decoder will also ignore the pin collected but not required for decoding.

Note: The output file will have **.pd* extension.

Note: The decoded file will be put into the specified path, or in the working directory in case the path does not exist.

Parameters

- **decoded_file** (*str*) – Name of the output file, which can be opened in text editor.
- **options** (*str*) – Additional options to be passed to sigrok-cli.

Returns

Return type None

display()

Draw digital waveforms in ipython notebook.

It utilises the wavedrom java script library, documentation for which can be found here: https://code.google.com/p/wavedrom/.

Note: Only use this method in Jupyter notebook.

Note: WaveDrom.js and WaveDromSkin.js are required under the subdirectory js.

Example of the data format to draw waveform:

```
>>> data = {'signal': [
```

{'name': 'clk', 'wave': 'p.....l...'},

{'name': 'dat', 'wave': 'x.345x|=.x', 'data': ['D','A','T','A']},

{'name': 'req', 'wave': '0.1..0|1.0'},

{},

{'name': 'ack', 'wave': '1.....l01.'}

]}

Returns

Return type None

parse(*parsed*, *start_pos*, *stop_pos*)

Parse the input data and generate a **.csv* file.

This method can be used along with the DMA. The input data is assumed to be 64-bit. The generated **.csv* file can be then used as the trace file.

Note: PMODA and PMODB are sharing the same trace buffer with different sets of pins, while ARDUINO has its own trace buffer.

Note: The parsed file will be put into the specified path, or in the working directory in case the path does not exist.

Parameters

- **parsed** (*str*) – Name of the parsed output file which can be opened in text editor.
- **start_pos** (*int*) – Starting sample number, no less than 1.
- **stop_pos** (*int*) – Stopping sample number, no more than the maximum number of samples.

Returns

Return type None

set_metadata()

Set metadata for the trace.

A **.sr* file directly generated from **.csv* will not have any metadata. This method helps to set the sample rate, probe names, etc.

Returns

Return type None

show()

Show information about the specified protocol.

This method will print out useful information about the protocol.

Returns

Return type None

sr2csv()

Translate the **.sr* file to **.csv* file.

The translated **.csv* files can be used for interactive plotting. **.csv* file is human readable, and can be opened using text editor.

Note: This method also removes the redundant header that is generated by sigrok.

Returns

Return type None

start(*timeout=10*)
Start the DMA to capture the traces.

If length is not specified, the maximum number of samples will be captured.

Parameters **timeout** (*int*) – The time in number of milliseconds to wait for DMA to be idle.

Returns

Return type None

stop()
Stop the DMA after capture is done.

Note: There is an internal timeout mechanism in the DMA class.

Returns

Return type None

pynq.drivers.usb_wifi module

class pynq.drivers.usb_wifi.**Usb_Wifi**
Bases: object

This class controls the usb dongle wifi connection.

The board is compatible with RALink RT5370 devices.

Note: Administrator rights are necessary to create network interface file

wifi_port
str

string identifier of the wireless network device

connect(*ssid*, *password*)
Make a new wireless connection.

This function kills the wireless connection and connect to a new one using network ssid and WPA passphrase. Wrong ssid or passphrase will reject the connection.

Parameters

- **ssid** (*str*) – Unique identifier of the wireless network
- **password** (*str*) – String WPA passphrase necessary to access the network

Returns

Return type None

gen_network_file(*ssid*, *password*)
Generate the network authentication file.

Generate the file from network SSID and WPA passphrase

Parameters

- **ssid** (*str*) – String unique identifier of the wireless network
- **password** (*str*) – String WPA passphrase necessary to access the network

Returns

Return type None

reset()
Shutdown the network connection.

This function shutdown the network connection and delete the interface file.

Returns

Return type None

pynq.drivers.video module

class pynq.drivers.video.**Frame**(*width*, *height*, *frame=None*)
Bases: object

This class exposes the bytearray of the video frame buffer.

Note: The maximum frame width is 1920, while the maximum frame height is 1080.

frame
bytearray

The bytearray of the video frame buffer.

width
int

The width of a frame.

height
int

The height of a frame.

__del__()
Delete the frame buffer.

Delete the frame buffer and free the memory only if the frame buffer is not empty.

Parameters **None** –

Returns

Return type None

__getitem__(*pixel*)
Get one pixel in a frame.

The pixel is accessed in the following way: *frame[x, y]* to get the tuple (r,g,b)

or *frame[x, y][rgb]* to access a specific color.

Examples

Get the three component of pixel (48,32) as a tuple, assuming the object is called *frame*:

```
>>> frame[48,32]
```

(128,64,12)

Access the green component of pixel (48,32):

```
>>> frame[48,32][1]
```

64

Note: The original frame stores pixels as (b,g,r). Hence, to return a tuple (r,g,b), we need to return (self.frame[offset+2], self.frame[offset+1], self.frame[offset]).

Parameters **pixel** (*list*) – A pixel (r,g,b) of a frame.

Returns A list of the current values (r,g,b) of the pixel.

Return type list

__setitem__(*pixel*, *value*)

Set one pixel in a frame.

The pixel is accessed in the following way: *frame[x, y] = (r,g,b)* to set the entire tuple

or *frame[x, y][rgb] = value* to set a specific color.

Examples

Set pixel (0,0), assuming the object is called *frame*:

```
>>> frame[0,0] = (255,255,255)
```

Set the blue component of pixel (0,0) to be 128

```
>>> frame[0,0][2] = 128
```

Note: The original frame stores pixels as (b,g,r).

Parameters

- **pixel** (*list*) – A pixel (r,g,b) of a frame.
- **value** (*list*) – A list of the values (r,g,b) to be set for the pixel.

Returns

Return type None

save_as_jpeg(*path*, *width=None*, *height=None*)
Save a video frame to a JPEG image.

Note: The JPEG filename must be included in the path.

Parameters

- **path** (*str*) – The path where the JPEG will be saved.
- **width** (*int*) – The width of the frame.
- **height** (*int*) – The height of the frame.

Returns

Return type None

class pynq.drivers.video.**HDMI**(*direction*, *video_mode=0*, *init_timeout=10*, *frame_list=None*, *vdma_name='SEG_axi_vdma_0_Reg'*, *display_name='SEG_v_tc_0_Reg'*, *capture_name='SEG_v_tc_1_Reg'*, *clk_name='SEG_axi_dynclk_0_reg0'*, *gpio_name='SEG_axi_gpio_video_Reg'*)
Bases: object

Class for an HDMI controller.

The frame buffer in an HDMI object can be shared among different objects. e.g., HDMI in and HDMI out objects can use the same frame buffer.

Note: HDMI supports direction 'in' and 'out'.

Examples

```
>>> hdmi = HDMI('in')
```

```
>>> hdmi = HDMI('out')
```

direction
str

Can be 'in' for HDMI IN or 'out' for HDMI OUT.

frame_list
_framebuffer

A frame buffer storing at most 3 frames.

__del__()
Delete the HDMI object.

Stop the video controller first to avoid odd behaviors of the DMA.

Returns

Return type None

frame = None
Wraps the raw version using the Frame object.

Use frame([index]) to read the frame more easily.

Parameters **index** (*int, optional*) – Index of the frames, from 0 to 2.

Returns A Frame object with accessible pixels.

Return type *Frame*

frame_addr = None
Get the current frame address.

Parameters **i** (*int, optional*) – Index of the current frame buffer.

Returns Address of the frame, thus current frame buffer.

Return type int

frame_height = None
Get the current frame height.

Parameters **None** –

Returns The height of the frame.

Return type int

frame_index = None
Get the frame index.

Use frame_index([new_frame_index]) to access the frame index. If *new_frame_index* is not specified, get the current frame index. If *new_frame_index* is specified, set the current frame to the new index.

Parameters **new_frame_index** (*int, optional*) – Index of the frames, from 0 to 2.

Returns The index of the active frame.

Return type int

frame_index_next = None
Change the frame index to the next one.

Parameters **None** –

Returns The index of the active frame.

Return type int

frame_phyaddr = None
Get the current physical frame address.

Parameters **i** (*int, optional*) – Index of the current frame buffer.

Returns Physical address of the frame, thus current frame buffer.

Return type int

frame_raw = None
Get the frame as a bytearray.

User may use frame([index]) to access the frame, which may introduce some overhead in rare cases. The method frame_raw([i]) is faster, but the parameter *i* has to be calculated manually.

Parameters **i** (*int, optional*) – A location in the bytearray.

Returns The frame in its raw bytearray form.

Return type bytearray

frame_width = None
Get the current frame width.

Parameters **None** –

Returns The width of the frame.

Return type int

mode = None
Change the resolution of the display.

Users can use mode(new_mode) to change the resolution. Specifically, with *new_mode* to be:

0 : '640x480, 60Hz'

1 : '800x600, 60Hz'

2 : '1280x720, 60Hz'

3 : '1280x1024, 60Hz'

4 : '1920x1080, 60Hz'

If *new_mode* is not specified, return the current mode.

Parameters **new_mode** (*int*) – A mode index from 0 to 4.

Returns The resolution of the display.

Return type str

Raises ValueError – If *new_mode* is out of range.

start (*timeout=20*)
Start the video controller.

Parameters **None** –

Returns

Return type None

state = None
Get the state of the device as an integer value.

Parameters **None** –

Returns The state 0 (DISCONNECTED), or 1 (STREAMING), or 2 (PAUSED).

Return type int

stop = None
Stop the video controller.

Parameters **None** –

Returns

Return type None

Submodules

pynq.general_const module

pynq.gpio module

class pynq.gpio.**GPIO**(*gpio_index*, *direction*)

Bases: object

Class to wrap Linux's GPIO Sysfs API.

This GPIO class does not handle PL I/O.

index

int

The index of the GPIO, starting from the GPIO base.

direction

str

Input/output direction of the GPIO.

path

str

The path of the GPIO device in the linux system.

__del__()

Delete a GPIO object.

Returns

Return type None

static get_gpio_base()

This method returns the GPIO base using Linux's GPIO Sysfs API.

This is a static method. To use:

```
>>> from pynq import GPIO
```

```
>>> gpio = GPIO.get_gpio_base()
```

Note: For path '/sys/class/gpio/gpiochip138/', this method returns 138.

Returns The GPIO index of the base.

Return type int

static get_gpio_pin(*gpio_user_index*)

This method returns a GPIO instance for PS GPIO pins.

Users only need to specify an index starting from 0; this static method will map this index to the correct Linux GPIO pin number.

Note: The GPIO pin number can be calculated using: GPIO pin number = GPIO base + GPIO offset + user index e.g. The GPIO base is 138, and pin 54 is the base GPIO offset. Then the Linux GPIO pin would be (138 + 54 + 0) = 192.

Parameters **gpio_user_index** (*int*) – The index specified by users, starting from 0.

Returns The Linux Sysfs GPIO pin number.

Return type int

read()
The method to read a value from the GPIO.

Returns An integer read from the GPIO

Return type int

write(*value*)
The method to write a value into the GPIO.

Parameters **value** (*int*) – An integer value, either 0 or 1

Returns

Return type None

pynq.mmio module

class pynq.mmio.**MMIO**(*base_addr*, *length=4*, *debug=False*)
Bases: object

This class exposes API for MMIO read and write.

virt_base
int

The address of the page for the MMIO base address.

virt_offset
int

The offset of the MMIO base address from the virt_base.

base_addr
int

The base address, not necessarily page aligned.

length
int

The length in bytes of the address range.

debug
bool

Turn on debug mode if it is True.

mmap_file
file

Underlying file object for MMIO mapping

mem
mmap

An mmap object created when mapping files to memory.

array
numpy.ndarray

A numpy view of the mapped range for efficient assignment

__del__()
Destructor to ensure mmap file is closed

read(*offset=0*, *length=4*)
The method to read data from MMIO.

Parameters

- **offset** (*int*) – The read offset from the MMIO base address.
- **length** (*int*) – The length of the data in bytes.

Returns A list of data read out from MMIO

Return type list

write(*offset*, *data*)
The method to write data to MMIO.

Parameters

- **offset** (*int*) – The write offset from the MMIO base address.
- **data** (*int / bytes*) – The integer(s) to be written into MMIO.

Returns

Return type None

pynq.pl module

class pynq.pl.**Bitstream**(*bitfile_name*)
Bases: *pynq.pl.PL*

This class instantiates a programmable logic bitstream.

bitfile_name
str

The absolute path of the bitstream.

timestamp
str

Timestamp when loading the bitstream. Format: year, month, day, hour, minute, second, microsecond

download()
The method to download the bitstream onto PL.

Note: The class variables held by the singleton PL will also be updated.

Returns

Return type None

class pynq.pl.**Overlay**(*bitfile_name*)
Bases: *pynq.pl.PL*

This class keeps track of a single bitstream's state and contents.

The overlay class holds the state of the bitstream and enables run-time protection of bindlings.

Our definition of overlay is: "post-bitstream configurable design". Hence, this class must expose configurability through content discovery and runtime protection.

This class stores four dictionaries: IP, GPIO, Interrupt Controller and Interrupt Pin dictionaries.

Each entry of the IP dictionary is a mapping: 'name' -> [address, range, state]

where name (str) is the key of the entry. address (int) is the base address of the IP. range (int) is the address range of the IP. state (str) is the state information about the IP.

Each entry of the GPIO dictionary is a mapping: 'name' -> [pin, state]

where name (str) is the key of the entry. pin (int) is the user index of the GPIO, starting from 0. state (str) is the state information about the GPIO.

Each entry in the Interrupt dictionaries are of the form 'name' -> [parent, number]

where name (str) is the name of the pin or the interrupt controller parent (str) is the name of the parent controller or '' if attached directly to the PS7 number (int) is the interrupt number attached to

bitfile_name
str

The absolute path of the bitstream.

bitstream
Bitstream

The corresponding bitstream object.

ip_dict
dict

The addressable IP instances on the overlay.

gpio_dict
dict

The dictionary storing the PS GPIO pins.

interrupt_controllers
dict

The dictionary containing all interrupt controllers

interrupt_pins
dict

The dictionary containing all interrupts in the design

download()
The method to download a bitstream onto PL.

Note: After the bitstream has been downloaded, the "timestamp" in PL will be updated. In addition, both of the IP and GPIO dictionaries on PL will be reset automatically.

Returns

Return type None

is_loaded()
This method checks whether a bitstream is loaded.

This method returns true if the loaded PL bitstream is same as this Overlay's member bitstream.

Returns True if bitstream is loaded.

Return type bool

load_ip_data(*ip_name*, *data*)
This method loads the data to the addressable IP.

Calls the method in the super class to load the data. This method can be used to program the IP. For example, users can use this method to load the program to the Microblaze processors on PL.

Note: The data is assumed to be in binary format (.bin). The data name will be stored as a state information in the IP dictionary.

Parameters

- **ip_name** (*str*) – The name of the addressable IP.
- **data** (*str*) – The absolute path of the data to be loaded.

Returns

Return type None

reset()
This function resets the IP and GPIO dictionaries of the overlay.

Note: This function should be used with caution. If the overlay is loaded, it also resets the IP and GPIO dictionaries in the PL.

Returns

Return type None

class pynq.pl.**PL**
Bases: object

Serves as a singleton for *Overlay* and *Bitstream* classes.

This class stores two dictionaries: IP dictionary and GPIO dictionary.

Each entry of the IP dictionary is a mapping: 'name' -> [address, range, state]

where name (str) is the key of the entry. address (int) is the base address of the IP. range (int) is the address range of the IP. state (str) is the state information about the IP.

Each entry of the GPIO dictionary is a mapping: 'name' -> [pin, state]

where name (str) is the key of the entry. pin (int) is the user index of the GPIO, starting from 0. state (str) is the state information about the GPIO.

The timestamp uses the following format: year, month, day, hour, minute, second, microsecond

bitfile_name
str

The absolute path of the bitstream currently on PL.

timestamp
str

Bitstream download timestamp.

ip_dict
dict

The dictionary storing addressable IP instances; can be empty.

gpio_dict
dict

The dictionary storing the PS GPIO pins.

classmethod client_request(*address='/home/xilinx/pynq/bitstream/.log'*, *key=b'xilinx'*)
Client connects to the PL server and receives the attributes.

This method should not be used by the users directly. To check open pipes in the system, use *lsof | grep <address>* and *kill -9 <pid>* to manually delete them.

Parameters

- **address** (*str*) – The filename on the file system.
- **key** (*bytes*) – The authentication key of connection.

Returns

Return type None

classmethod load_ip_data(*ip_name*, *data*)
This method writes data to the addressable IP.

Note: The data is assumed to be in binary format (.bin). The data name will be stored as a state information in the IP dictionary.

Parameters

- **ip_name** (*str*) – The name of the addressable IP.
- **data** (*str*) – The absolute path of the data to be loaded.

Returns

Return type None

classmethod reset()
Reset both the IP and GPIO dictionaries.

This method must be called after a bitstream download. 1. In case there is a **.tcl* file, this method will reset the IP, Interrupt and GPIO dictionaries based on the tcl file. 2. In case there is no **.tcl* file, this method will simply clear the state information stored for all dictionaries.

classmethod server_update(*continued=1*)
Client sends the attributes to the server.

This method should not be used by the users directly. To check open pipes in the system, use *lsof | grep <address>* and *kill -9 <pid>* to manually delete them.

Parameters **continued** (*int*) – Continue (1) or stop (0) the PL server.

Returns

Return type None

classmethod setup(*address='/home/xilinx/pynq/bitstream/.log'*, *key=b'xilinx'*)

Start the PL server and accept client connections.

This method should not be used by the users directly. To check open pipes in the system, use *lsof | grep <address>* and *kill -9 <pid>* to manually delete them.

Parameters

- **address** (*str*) – The filename on the file system.
- **key** (*bytes*) – The authentication key of connection.

Returns

Return type None

class pynq.pl.**PL_Meta**

Bases: type

This method is the meta class for the PL.

This is not a class for users. Hence there is no attribute or method exposed to users.

bitfile_name

The getter for the attribute *bitfile_name*.

Returns The absolute path of the bitstream currently on PL.

Return type str

gpio_dict

The getter for the attribute *gpio_dict*.

Returns The dictionary storing the PS GPIO pins.

Return type dict

interrupt_controllers

The getter for the attribute *interrupt_controllers*

Returns The dictionary storing interrupt controller information

Return type dict

interrupt_pins

The getter for the attribute *interrupt_pins*

Returns The dictionary storing the interrupt endpoint information

Return type dict

ip_dict

The getter for the attribute *ip_dict*.

Returns The dictionary storing addressable IP instances; can be empty.

Return type dict

timestamp
The getter for the attribute *timestamp*.

Returns Bitstream download timestamp.

Return type str

pynq.xlnk module

class pynq.xlnk.**Xlnk**
Bases: object

Class to enable CMA memory management.

The CMA state maintained by this class is local to the application except for the *CMA Memory Available* attribute which is global across all the applications.

bufmap
dict

Mapping of allocated memory to the buffer sizes in bytes.

__del__()
Destructor for the current Xlnk object.

Frees up all the memory which was allocated through current object.

Returns

Return type None

cma_alloc(*length*, *cacheable=0*, *data_type='void'*)
Allocate physically contiguous memory buffer.

Allocates a new buffer and adds it to *bufmap*.

Possible values for parameter *cacheable* are:

1: the memory buffer is cacheable.

0: the memory buffer is non-cacheable.

Examples

mmu = Xlnk()

Allocate 10 *void* * memory locations.

m1 = mmu.cma_alloc(10)

Allocate 10 *float* * memory locations.

m2 = mmu.cma_alloc(10, data_type = "float")

Notes

1. Total size of buffer is automatically calculated as size = length * sizeof(data_type)

2. This buffer is allocated inside the kernel space using xlnk driver. The maximum allocatable memory is defined at kernel build time using the CMA memory parameters. For Pynq-Z1 kernel, it is specified as 128MB.

The unit of *length* depends upon the *data_type* argument.

Parameters

- **length** (*int*) – Length of the allocated buffer. Default unit is bytes.
- **cacheable** (*int*) – Indicating whether or not the memory buffer is cacheable.
- **data_type** (*str*) – CData type of the allocated buffer. Should be a valid C-Type.

Returns An CFFI object which can be accessed similar to arrays.

Return type cffi.FFI.CData

static cma_cast(*data*, *data_type='void'*)
Cast underlying buffer to a specific C-Type.

Input buffer should be a valid object which was allocated through *cma_alloc* or a CFFI pointer to a memory buffer. Handy for changing void buffers to user defined buffers.

Parameters

- **data** (*cffi.FFI.CData*) – A valid buffer pointer allocated via *cma_alloc*.
- **data_type** (*str*) – New data type of the underlying buffer.

Returns Pointer to buffer with specified data type.

Return type cffi.FFI.CData

cma_free(*buf*)
Free a previously allocated buffer.

Input buffer should be a valid object which was allocated through *cma_alloc* or a CFFI pointer to a memory buffer.

Parameters **buf** (*cffi.FFI.CData*) – A valid buffer pointer allocated via *cma_alloc*.

Returns

Return type None

cma_get_buffer(*buf*, *length*)
Get a buffer object.

Used to get an object which supports python buffer interface. The return value thus, can be cast to objects like *bytearray*, *memoryview* etc.

Parameters

- **buf** (*cffi.FFI.CData*) – A valid buffer object which was allocated through *cma_alloc*.
- **length** (*int*) – Length of buffer in Bytes.

Returns A CFFI object which supports buffer interface.

Return type cffi.FFI.CData

cma_get_phy_addr(*buf_ptr*)
Get the physical address of a buffer.

Used to get the physical address of a memory buffer allocated with *cma_alloc*. The return value can be used to access the buffer from the programmable logic.

Parameters **buf_ptr** (*cffi.FFI.CData*) – A void pointer pointing to the memory buffer.

Returns The physical address of the memory buffer.

Return type int

static cma_memcopy(*dest*, *src*, *nbytes*)

High speed memcopy between buffers.

Used to perform a byte level copy of data from source buffer to the destination buffer.

Parameters

- **dest** (*cffi.FFI.CData*) – Destination buffer object which was allocated through *cma_alloc*.
- **src** (*cffi.FFI.CData*) – Source buffer object which was allocated through *cma_alloc*.
- **nbytes** (*int*) – Number of bytes to copy.

Returns

Return type None

cma_stats()

Get current CMA memory Stats.

CMA Memory Available : Systemwide CMA memory availability.

CMA Memory Usage : CMA memory used by current object.

Buffer Count : Buffers allocated by current object.

Returns Dictionary of current stats.

Return type dict

xlnk_reset()

Systemwide Xlnk Reset.

Notes

This method resets all the CMA buffers allocated across the system.

Returns

Return type None

pynq.xlnk.**sig_handler**(*signum*, *frame*)

CHAPTER 19

Verification

Table of Contents

This section documents the test infrastructure supplied with the *pynq* package. It is organized as follows:

- *Running Tests* : describes how to run the pytest.
- *Writing Tests* : explains how to write tests.
- *Miscellaneous* : covers additional information relating to tests.

Running Tests

The *pynq* package provides tests for most python modules.

To run all the tests together, pytest can be run in a Linux terminal on the board. All the tests will be automatically collected in the current directory and child directories.

Note: The pytests have to be run as root

```
cd /home/xilinx/pynq
sudo py.test -vsrw
```

For a complete list of pytest options, please refer to Usage and Invocations - Pytest.

Collection Phase

During this phase, the pytest will collect all the test modules in the current directory and all of its child directories. The user will be asked if a Pmod is connected, and to which port it is connected.

For example:

```
Pmod OLED attached to the board? ([yes]/no)>>> yes
Type in the interface ID of the Pmod OLED (A or B):
```

For the answer to the first question, “yes”, “YES”, “Yes”, “y”, and “Y” are acceptable; the same applies for “no” as an answer. You can also press *Enter*; this is equivalent to “yes”.

Type “A” (for PMODA) or “B” (for PMODB).

Answering “No” will skip the corresponding test(s) during the testing phase.

Testing Phase

The test suite will guide the user through all the tests implemented in the pynq package. As part of the tests, the user will be prompted for confirmation that the tests have passed, for example:

```
test_led0 ...
Onboard LED 0 on? ([yes]/no)>>>
```

Again press “Enter”, or type “yes”, “no” etc.

At the end of the testing phase, a summary will be given to show users how many tests are passed / skipped / failed.

Writing Tests

This section follows the guide available on Pytest Usages and Examples. You can write a test class with assertions on inputs and outputs to allow automatic testing. The names of the test modules *must* start with *test_*; all the methods for tests in any test module *must* also begin with *test_*. One reason to enforce this is to ensure the tests will be collected properly. See the Full pytest documentation for more details.

Step 1

First of all, the pytest package has to be imported:

```
import pytest
```

Step 2

Decorators can be specified directly above the methods. For example, you can specify (1) the order of this test in the entire pytest process, and (2) the condition to skip the corresponding test. More information on decorators can be found in Marking test functions with attributes - Pytest.

```
@pytest.mark.run(order=26)
@pytest.mark.skipif(not flag, reason="need both ADC and DAC attached")
```

Step 3

Directly below decorators, you can write some assertions/tests. See the example below:

```
@pytest.mark.run(order=26)
@pytest.mark.skipif(not flag, reason="need both ADC and DAC attached")
def test_loop_single():
    """Test for writing a single value via the loop.

    First check whether read() correctly returns a string. Then ask the users
    to write a voltage on the DAC, read from the ADC, and compares the two
```

```
    voltages.

    The exception is raised when the difference is more than 10% and more than
    0.1V.

    Note
    ----
    Users can use a straight cable (instead of wires) to do this test.
    For the 6-pin DAC Pmod, it has to be plugged into the upper row of the
    Pmod interface.

    """
    global dac,adc
    dac = Pmod_DAC(dac_id)
    adc = Pmod_ADC(adc_id)

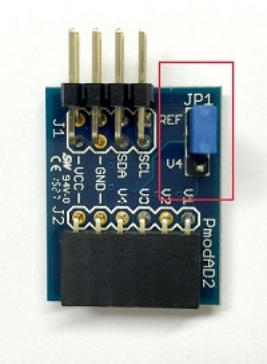
    value = float(input("\nInsert a voltage in the range of [0.00, 2.00]: "))
    assert value<=2.00, 'Input voltage should not be higher than 2.00V.'
    assert value>=0.00, 'Input voltage should not be lower than 0.00V.'
    dac.write(value)
    sleep(0.05)
    assert round(abs(value-adc.read()[0]),2)<max(0.1, 0.1*value), \
            'Read value != write value.'
```

Note the *assert* statements specify the desired condition, and raise exceptions whenever that condition is not met. A customized exception message can be attached at the end of the *assert* methods, as shown in the example above.

Miscellaneous Test Setup

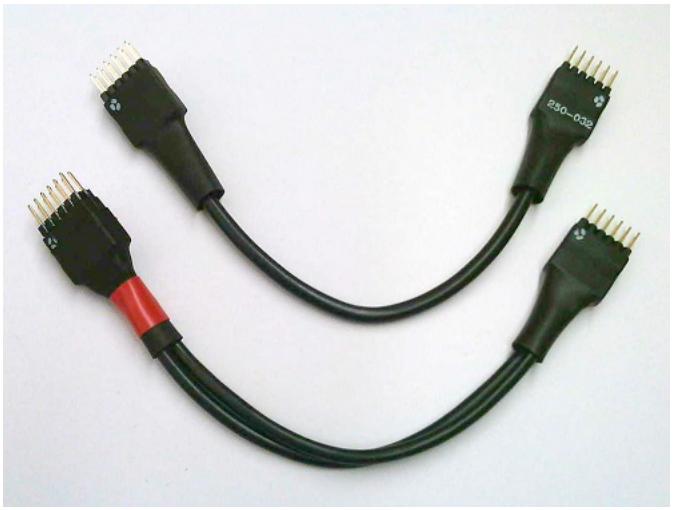
ADC Jumper

In our tests and demos, we have used a Pmod ADC. In order to make it work properly with the testing environment, you need to set a jumper **JP1** to **REF** on the Pmod ADC. This will allow the ADC to use the correct reference voltage.



Cable Type

Two types of cables can be used with the tests in the pynq package, a “straight” cable, and a “loopback” cable:



- *Straight cable* (upper one in the image): The internal wires between the two ends are straight. This cable is intended for use as an extension cable.
- *Loopback cable* (lower one in the image, with red ribbon): The internal wires are twisted. This cable is intended for testing.

There are marks on the connectors at each end of the cable to indicate the orientation and wiring of the cable.

Note: You must not short VCC and GND as it may damage the board. It is good practice to align the pins with the dot marks to VCC of the Pmod interfaces.

Note: For testing, there is only one connection type (mapping) allowed for each cable type. Otherwise VCC and GND could be shorted, damaging the board.

CHAPTER 20

Frequently Asked Questions (FAQs)

Connecting to the board

I can't connect to my board

1. Check the board is powered on (Red LED LD13) and that the bitstream has been loaded (Green "DONE" LED LD12)
2. Your board and PC/laptop must be on the same network, or have a direct network connection. Check that you can *ping* the board (hostname, or IP address) from a command prompt or terminal on your host PC

```
>ping pynq
```

or

```
>ping 192.168.2.99
```

   (The default IP address of the board is : 192.168.2.99)

3. Log on to the board through a terminal, and check the system is running. i.e. that the Linux shell is accessible. See below for details on logging on with a terminal.
4. From a terminal, check you can ping the board from your host PC, and also ping your host PC from the board

   If you can't ping the board, or the host PC, check your network settings.

   - You must ensure board your PC and board are connected to the same network, and have IP addresses in the same range. If your network cables is connected directly to your PC/laptop and board, you may need to set a static IP address for your PC/laptop manually. See the Appendix: Assign your PC/Laptop a static ip address
   - If you have a proxy setup, you may need to add a rule to bypass the board hostname/ip address.
   - If you are using a docking station, when your laptop is docked, the Ethernet port on the PC may be disabled.

My board is not powering on (No Red LED)

The board can be powered by USB cable, or power adapter (7 - 15V V 2.1mm centre-positive barrel jack). Make sure Jumper JP5 is set to USB or REG (for power adapter). If powering the board via USB, make sure the USB port is fully powered. Laptops in low power mode may reduce the available power to a USB port.

The bitstream is not loading (No Green LED)

- Check the Micro-SD card is inserted correctly (the socket is spring loaded, so push it in until you feel it click into place).
- Check jumper JP4 is set to SD (board boots from Micro SD card).
- Connect a terminal and verify that the Linux boot starts.

If the Linux boot does not start, or fails, you may need to flash the Micro SD card with the PYNQ-Z1 image.

The hostname of the board is not resolving/not found

It may take the hostname (pynq) some time to resolve on your network. If you know the IP address of the board, it may be faster to use the IP address to navigate to the Jupyter portal instead of the hostname.

e.g. In your browser, go to:

```
http://192.168.2.99:9090
```

You need to know the IP address of the board first. You can find the IP by connecting a terminal to the board. You can run *ifconfig* in the Linux shell on the board to check the network settings. Check the settings for *eth0* and look for an IP address.

I don't have an Ethernet port on my PC/Laptop

If you don't have an Ethernet port, you can get a USB to Ethernet adapter.

If you have a wireless router with Ethernet ports (LAN), you can connect your PYNQ-Z1 board to an Ethernet port on your router, and connect to it from your PC using WiFi. (You may need to change settings on your Router to enable the Wireless network to communicate with your LAN - check your equipment documentation for details.)

You can also connect a WiFi dongle to the board, and set up the board to connect to the wireless network. Your host PC can then connect to the same wireless network to connect to the board.

How do I setup my computer to connect to the board?

If you are connecting your board to your network (i.e. you have plugged the Ethernet cable into the board, and the other end into a network switch, or home router), then you should not need to setup anything on your computer. Usually, both your computer, and board will be assigned an IP address automatically, and they will be able to communicate with each other.

If you connect your board directly to your computer with an ethernet cable, then you need to make sure that they have IP addresses in the same range. The board will assign itself a static IP address (by default 192.168.2.99), and you will need to assign a static IP address in the same range to the computer. This allows your computer and board to communicate to each other over the Ethernet cable.

See the Appendix: Assign your PC/Laptop a static ip address

I can't connect to the Jupyter portal

My Board is powered on, and I see the Red and Green LEDs, but I can't connect to the Jupyter Portal, or see the Samba shared drive:

By default, the board has DHCP enabled. If you plug the board into a home router, or network switch connected to your network, it should be allocated an IP address automatically. If not, it should fall back to a static IP address of *192.168.2.99*

If you plug the Ethernet cable directly to your computer, you will need to configure your network card to have an IP in the same address range. e.g. *192.168.2.1*

My board is connected, and I have verified the IP addresses on the board and my network interface, but I cannot connect to the board.

VPN

If your PC/laptop is connected to a VPN, and your board is not on the same VPN network, this will block access to local IP addresses. You need to disable the VPN, or set it to bypass the board address.

Proxy

If your board is connected to a network that uses a proxy, you need to set the proxy variables on the board

```
set http_proxy=my_http_proxy:8080
set https_proxy=my_https_proxy:8080
```

How do I connect to the board using a terminal?

To do this, you need to connect to the board using a terminal.

To connect a terminal: Connect a Micro USB cable to the board and your PC/Laptop, and use a terminal emulator (puTTY, TeraTerm etc) to connect to the board.

Terminal Settings:

- 115200 baud
- 8 data bits
- 1 stop bit
- No Parity
- No Flow Control

Once you connect to the board, you can configure the network interface in Linux

Board/Jupyter settings

How do I modify the board settings?

Linux is installed on the board. Connect to the board using a terminal, and change the settings as you would for any other Linux machine.

How do I find the IP address of the board?

Connect to the board using a terminal (see above) and type 'hostname -I' to find the IP address for the eth0 Ethernet adapter or the WiFi dongle.

How do I set/change the static IP address on the board?

The Static IP address is set in `/etc/dhcp/dhclient.conf` - you can modify the board's static IP here.

How do I find my hostname?

Connect to the board using a terminal and run `hostname`

How do I change the hostname?

If you have multiple boards on the same network, you should give them different host names. You can run the following script to change the hostname:

```
sudo /home/xilinx/scripts/hostname.sh NEW_HOST_NAME
```

What is the user account and password?

Username and password for all Linux, jupyter and samba logins are: `xilinx/xilinx`

I can't log in to the Jupyter portal with Safari on Mac OS

This is a known issue with Safari and is related to Safari not authenticating the Jupyter password properly. To workaround, you can use another browser, or disable the password

How do I enable/disable the Jupyter notebook password

The Jupyter configuration file can be found at

```
/root/.jupyter/jupyter_notebook_config.py
```

You can add or comment out the c.NotebookApp.password to bypass the password authentication when connecting to the Jupyter Portal.

```
c.NotebookApp.password =u'sha1:6c2164fc2b22:ed55ecf07fc0f985ab46561483c0e888e8964ae6'
```

How do I change the Jupyter notebook password

A hashed password is saved in the Jupyter Notebook configuration file.

```
/root/.jupyter/jupyter_notebook_config.py
```

You can create a hashed password using the function *IPython.lib.passwd()*:

```
from IPython.lib import passwd
password = passwd("secret")
6c2164fc2b22:ed55ecf07fc0f985ab46561483c0e888e8964ae6
```

You can then add or modify the line in the *jupyter_notebook_config.py* file

```
c.NotebookApp.password =u'sha1:6c2164fc2b22:ed55ecf07fc0f985ab46561483c0e888e8964ae6'
```

General Questions

Does Pynq support Python 2.7?

Python 2.7 is loaded on Zynq® and Python 2.7 scripts can be executed. Pynq, however, is based on Python 3.4. No attempts have been made to ensure backward compatibility with Python 2.7.

Where can I get the PYNQ-Z1 image?

You can Download the PYNQ-Z1 image here

How do I write the Micro SD card image

You can find instructions in the Appendix: Writing the SD card image

What type of Micro SD card do I need?

We recommend you use a card at least 8GB in size and at least class 4 speed rating.

CHAPTER 21

Glossary

Table of Contents

A-G

APSOC All Programmable System on Chip

BSP A board support package (BSP) is a collection of low-level libraries and drivers. The Xilinx® software development Kit (SDK) uses a BSP to form the lowest layer of your application software stack. Software applications must link against or run on top of a given software platform using the APIs that it provides. Therefore, before you can create and use software applications in SDK, you must create a board support package

FPGA Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing. This feature distinguishes FPGAs from Application Specific Integrated Circuits (ASICs), which are custom manufactured for specific design tasks.

H-R

Hardware Platform An SDK project

HDF Hardware Definition File (.hdf). This file is created by Vivado and contains information about a processor system in an FPGA overlay. The HDF specifies the peripherals that exist in the system, and the memory map. This is used by the BSP to build software libraries to support the available peripherals.

I2C See IIC

IIC Inter-Integrated Circuit; multi-master, multi-slave, single-ended, serial computer bus protocol

IOPs Input/Output Processors

Jupyter (Notebooks) Jupyter is an open source project consisting of an interactive, web application that allows users to create and share notebook documents that contain live code and the full range of rich media supported by modern browsers. These include text, images, videos, LaTeX-styled equations, and interactive widgets. The Jupyter framework is used as a front-end to over 40 different programming languages. It originated from the interactive data science and scientific computing communities. Its uses include: data cleaning and transformation, numerical simulation, statistical modelling, machine learning and much more.

MicroBlaze MicroBlaze is a soft microprocessor core designed for Xilinx FPGAs. As a soft-core processor, MicroBlaze is implemented entirely in the general-purpose memory and logic fabric of an FPGA.

Pmod Interface The Pmod or Peripheral Module interface is used to connect low frequency, low I/O pin count peripheral modules to host controller boards.accessory boards to add functionality to the platform. e.g. ADC, DAC, I/O interfaces, sensors etc.

(Micro) SD Secure Digital (Memory Card standard)

readthedocs.org readthedocs.org is a popular website that hosts the documentation for open source projects at no cost. readthedocs.org uses Sphinx document generation tools to automatically generate both the website and PDF versions of project documentation from a GitHub repository when new updates are pushed to that site.

REPL A read–eval–print loop (REPL), also known as an interactive toplevel or language shell, is a simple, interactive computer programming environment that takes single user inputs (i.e. single expressions), evaluates them, and returns the result to the user; a program written in a REPL environment is executed piecewise. The term is most usually used to refer to programming interfaces similar to the classic Lisp machine interactive environment. Common examples include command line shells and similar environments for programming languages, and is particularly characteristic of scripting languages wikipedia

reST Restructured text is a markup language used extensively with the Sphinx document generator

S-Z

SDK Xilinx SDK - Software Development Kit. Software development environment including cross-compiles for ARM®, and MicroBlaze processors. Also includes debug, and profiling tools. Required to build software for a MicroBlaze processor inside an IOP.

SOC System On Chip

Sphinx A document generator written in Python and used extensively to document Python and other coding projects

SPI Serial Peripheral Interface; synchronous serial communication interface specification

UART Universal asynchronous receiver/transmitter; Serial communication protocol

Vivado Vivado Design Suite is a suite of computer-aided design tools provided by Xilinx for creating FPGA designs. It is used to design and implement the overlays used in Pynq.

XADC An XADC is a hard IP block that consists of dual 12-bit, 1 Mega sample per second (MSPS), analog-to-digital converters and on-chip sensors which are integrated into Xilinx 7 series FPGA devices

Zynq® Zynq-7000 All Programmable SoC (APSoC) devices integrate the software programmability of an ARM®-based processor with the hardware programmability of an FPGA, enabling key analytics and hardware acceleration while integrating CPU, DSP, ASSP, and mixed signal functionality on a

single device. Zynq-7000 AP SoCs infuse customizable intelligence into today's embedded systems to suit your unique application requirements

Zynq PL Programmable Logic - FPGA fabric

Zynq PS Processing System - SOC processing subsystem built around dual-core, ARM Cortex-A9 processor

CHAPTER 22

Useful Reference Links

Table of Contents

Git

- Interactive introduction to Git
- Free PDF copy of The Pro Git book by Scott Chacon and Ben Straub

Jupyter

- Jupyter Project
- Try Jupyter in your browser

PUTTY (terminal emulation software)

- PUTTY download page

Pynq Technical support

- Pynq Technical support

Python built-in functions

- C Python native functions

Python training

- The Python Tutorial from the Python development team
- Google Python training including videos
- Python Tutor including visualization of Python program execution
- 20 Best Free Tutorials to Learn Python as of 9 Oct 2015

reStructuredText

- reStructuredText docs
- reStructuredText Primer
- Online reStructuredText editor

Sphinx

- The official Sphinx docs
- Online reST and Sphinx editor with rendering
- A useful Sphinx cheat sheet
- Jupyter Notebook Tools for Sphinx

CHAPTER 23

Appendix

Table of Contents

Technology Backgrounder

Overlays and Design Re-use

The 'magic' of mapping an application to an APSoC, without designing custom hardware, is achieved by using *FPGA overlays*. FPGA overlays are FPGA designs that are both highly configurable and highly optimized for a given domain. The availability of a suitable overlay removes the need for a software designer to develop a new bitstream. Software and system designers can customize the functionality of an existing overlay *in software* once the API for the overlay bitstream is available.

An FPGA overlay is a domain-specific FPGA design that has been created to be highly configurable so that it can be used in as many different applications as possible. It has been crafted to maximize post-bitstream programmability which is exposed via its API. The API provides a new entry-point for application-focused software and systems engineers to exploit APSoCs in their solutions. With an API they only have to write software to program configure the functions of an overlay for their applications.

By analogy with the Linux kernel and device drivers, FPGA overlays are designed by relatively few engineers so that they can be re-used by many others. In this way, a relatively small number of overlay designers can support a much larger community of APSoC designers. Overlays exist to promote re-use. Like kernels and device drivers, these hardware-level artefacts are not static, but evolve and improve over time.

Characteristics of Good Overlays

Creating one FPGA design and its corresponding API to serve the needs of many applications in a given domain is what defines a successful overlay. This, one-to-many relationship between the overlay and its users, is different from the more common one-to-one mapping between a bitstream and its application.

Consider the example of an overlay created for controlling drones. Instead of creating a design that is optimized for controlling just a single type of drone, the hardware architects recognize the many common requirements shared by different drone controllers. They create a design for controlling drones that is a flexible enough to be used with

several different drones. In effect, they create a drone-control overlay. They expose, to the users of their bitstream, an API through which the users can determine in software the parameters critical to their application. For example, a drone control overlay might support up to eight, pulse-width-modulated (PWM) motor control circuits. The software programmer can determine how many of the control circuits to enable and how to tune the individual motor control parameters to the needs of his particular drone.

The design of a good overlay shares many common elements with the design of a class in object-oriented software. Determining the fundamental data structure, the private methods and the public interface are common requirements. The quality of the class is determined both by its overall usefulness and the coherency of the API it exposes. Well-engineered classes and overlays are inherently useful and are easy to learn and deploy.

Pynq adopts a holistic approach by considering equally the design of the overlays, the APIs exported by the overlays, and how well these APIs interact with new and existing Python design patterns and idioms to simplify and improve the APSoC design process. One of the key challenges is to identify and refine good abstractions. The goal is to find abstractions that improve design coherency by exposing commonality, even among loosely-related tasks. As new overlays and APIs are published, we expect that the open-source software community will further improve and extend them in new and unexpected ways.

Note that FPGA overlays are not a novel concept. They have been studied for over a decade and many academic papers have been published on the topic.

The Case for Productivity-layer Languages

Successive generations of All Programmable Systems on Chip embed more processors and greater processing power. As larger applications are integrated into APSoCs, the embedded code increases also. Embedded code that is speed or size critical, will continue to be written in C/C++. These 'efficiency-layer or systems languages' are needed to write fast, low-level drivers, for example. However, the proportion of embedded code that is neither speed-critical or size-critical, is increasing more rapidly. We refer to this broad class of code as *embedded applications code*.

Programming embedded applications code in higher-level, 'productivity-layer languages' makes good sense. It simply extends the generally-accepted best-practice of always programming at the highest possible level of abstraction. Python is currently a premier productivity-layer language. It is now available in different variants for a range of embedded systems, hence its adoption in Pynq. Pynq runs CPython on Linux on the ARM® processors in Zynq® devices. To further increase productivity and portability, Pynq uses the Jupyter Notebook, an open-source web framework to rapidly develop systems, document their behavior and disseminate the results.

Writing the SD card image

Windows

- Insert the Micro SD card into your SD card reader and check which drive letter was assigned. You can find this by opening Computer/My Computer in Windows Explorer.
- Download the Win32DiskImager utility from the Sourceforge Project page
- Extract the *Win32DiskImager* executable from the zip file and run the Win32DiskImager utility as administrator. (Right-click on the file, and select Run as administrator.)
- Select the PYNQ-Z1 image file (.img).
- Select the drive letter of the SD card. Be careful to select the correct drive. If you select the wrong drive you can overwrite data on that drive. This could be another USB stick, or memory card connected to your computer, or your computer's hard disk.
- Click **Write** and wait for the write to complete.

MAC

On Mac OS, you can use dd, or the graphical tool ImageWriter to write to your Micro SD card.

- First open a terminal and unzip the image:

```
unzip pynq_z1_image_2016_09_14.zip -d ./
```

ImageWriter

Note the Micro SD card must be formatted as FAT32.

- Insert the Micro SD card into your SD card reader
- From the Apple menu, choose "About This Mac", then click on "More info..."; if you are using Mac OS X 10.8.x Mountain Lion or newer, then click on "System Report".
- Click on "USB" (or "Card Reader" if using a built-in SD card reader) then search for your SD card in the upper-right section of the window. Click on it, then search for the BSD name in the lower-right section; it will look something like **diskn** where n is a number (for example, disk4). Make sure you take a note of this number.
- Unmount the partition so that you will be allowed to overwrite the disk. To do this, open Disk Utility and unmount it; do not eject it, or you will have to reconnect it. Note that on Mac OS X 10.8.x Mountain Lion, "Verify Disk" (before unmounting) will display the BSD name as */dev/disk1s1* or similar, allowing you to skip the previous two steps.
- From the terminal, run the following command:

```
sudo dd bs=1m if=path_of_your_image.img of=/dev/rdiskn
```

Remember to replace n with the number that you noted before!

If this command fails, try using disk instead of rdisk:

```
sudo dd bs=1m if=path_of_your_image.img of=/dev/diskn
```

Wait for the card to be written. This may take some time.

Command Line

- Open a terminal, then run:

```
diskutil list
```

- Identify the disk (not partition) of your SD card e.g. disk4, not disk4s1.
- Unmount your SD card by using the disk identifier, to prepare for copying data to it:

```
diskutil unmountDisk /dev/disk<disk# from diskutil>
```

  where disk is your BSD name e.g. *diskutil unmountDisk /dev/disk4*

- Copy the data to your SD card:

```
sudo dd bs=1m if=image.img of=/dev/rdisk<disk# from diskutil>
```

  where disk is your BSD name e.g. sudo dd bs=1m if=pynq_z1_image_2016_09_07.img of=/dev/rdisk4

This may result in a dd: invalid number '1m' error if you have GNU coreutils installed. In that case, you need to use a block size of 1M in the bs= section, as follows:

```
sudo dd bs=1M if=image.img of=/dev/rdisk<disk# from diskutil>
```

Wait for the card to be written. This may take some time. You can check the progress by sending a SIGINFO signal (press Ctrl+T).

If this command still fails, try using disk instead of rdisk, for example:

```
sudo dd bs=1m if=pynq_z1_image_2016_09_07.img of=/dev/disk4
```

Linux

dd

Please note the dd tool can overwrite any partition on your machine. Please be careful when specifying the drive in the instructions below. If you select the wrong drive, you could lose data from, or delete your primary Linux partition.

- Run *df -h* to see what devices are currently mounted.
- Insert the Micro SD card into your SD card reader
- Run df -h again.

The new device that has appeared is your Micro SD card. The left column gives the device name; it will be listed as something like /dev/mmcblk0p1 or /dev/sdd1. The last part (p1 or 1 respectively) is the partition number but you want to write to the whole SD card, not just one partition. You need to remove that part from the name. e.g. Use /dev/mmcblk0 or /dev/sdd as the device name for the whole SD card.

Now that you've noted what the device name is, you need to unmount it so that files can't be read or written to the SD card while you are copying over the SD image.

- Run *umount /dev/sdd1*, replacing sdd1 with whatever your SD card's device name is (including the partition number).

If your SD card shows up more than once in the output of df due to having multiple partitions on the SD card, you should unmount all of these partitions.

- In the terminal, write the image to the card with the command below, making sure you replace the input file if= argument with the path to your .img file, and the /dev/sdd in the output file of= argument with the right device name. This is very important, as you will lose all data on the hard drive if you provide the wrong device name. Make sure the device name is the name of the whole Micro SD card as described above, not just a partition of it; for example, sdd, not sdds1, and mmcblk0, not mmcblk0p1.

sudo dd bs=4M if=pynq_z1_image_2016_09_07.img of=/dev/sdd

Please note that block size set to 4M will work most of the time; if not, please try 1M, although this will take considerably longer.

The dd command does not give any information of its progress and so may appear to have frozen; it could take a few minutes to finish writing to the card.

Instead of dd you can use *dcfldd*; it will give a progress report about how much has been written.

Assign your laptop/PC a static IP address

Instructions may vary slightly depending on the version of operating system you have. You can also search on google for instructions on how to change your network settings.

You need to set the IP address of your laptop/pc to be in the same range as the board. e.g. if the board is 192.168.2.99, the laptop/PC can be 192.168.2.x where x is 0-255 (excluding 99, as this is already taken by the board).

You should record your original settings, in case you need to revert to them when finished using PYNQ.

Windows

- Go to Control Panel -> Network and Internet -> Network Connections
- Find your Ethernet network interface, usually *Local Area Connection*
- Double click on the network interface to open it, and click on *Properties*
- Select Internet Protocol Version 4 (TCP/IPv4) and click *Properties*
- Select *Use the following IP address*
- Set the Ip address to 192.168.2.1 (or any other address in the same range as the board)
- Set the subnet mask to 255.255.255.0 and click **OK**

Mac OS

OS X

- Open *System Preferences* then open *Network*
- Click on the connection you want to set manually, usually *Ethernet*
- From the Configure IPv4 drop down choose Manually
- Set the IP address to 192.168.2.1 (or any other address in the same range as the board)
- Set the subnet mask to 255.255.255.0 and click **OK**

The other settings can be left blank.

Linux

- Edit this file (replace gedit with your preferred text editor):

  sudo gedit /etc/network/interfaces

The file usually looks like this:

```
auto lo eth0
iface lo inet loopback
iface eth0 inet dynamic
```

- Make the following change to set the eth0 interface to the static IP address 192.168.2.1

```
iface eth0 inet static
   address 192.168.2.1
   netmask 255.255.255.0
```

Your file should look like this:

```
    auto lo eth0
    iface lo inet loopback
    iface eth0 inet static
            address 192.168.2.1
            netmask 255.255.255.0
```

CHAPTER 24

Change log

Table of Contents

Version 1.4

Image release: pynq_z1_image_2016_02_10

Documentation updated: 10 Feb 2017

- Xilinx Linux kernel upgraded to 4.6.0
- **Added Linux Packages**
  - Python3.6
  - iwconfig
  - iwlist
  - microblaze-gcc
- **New Python Packages**
  - asyncio
  - uvloop
  - transitions
  - pygraphviz
  - pyeda
- **Updated Python Pacakges**
  - pynq
  - Jupyter Notebook Extension added
  - IPython upgraded to support Python 3.6
  - pip

- **Other changes**
  - Jupyter extensions
  - reveal.js updated
  - update_pynq.sh
  - wavedrom.js
- **Base overlay changes**
  - IOP interface to DDR added (Pmod and Arduino IOP)
  - Interrupt controller from overlay to PS added. IOP GPIO connected to Interrupt controller.
  - Arduino GPIO base address has changed due to merge of GPIO into a single block. *arduino_grove_ledbar* and *arduino_grove_buzzer* compiled binaries are not backward compatible with previous Pynq overlay/image.
- **Pynq API/driver changes**
  - TraceBuffer: Bit masks are not required. Only pins should be specified.
  - PL: `pl_dict` returns an integer type for any base address / address range.
  - Video: Video mode constants are exposed outside the class.
  - Microblaze binaries for IOP updated.
  - Xlnk driver updated to support SDx 2016.3. Removed the customized Xlnk drivers and use the libsds version.
- **Added new iop modules**
  - arduino_lcd18
- **Added Notebooks**
  - audio_playback (updated)
  - arduino_lcd18 (new)
  - asyncio_buttons (new)
- **Documentation changes**
  - New section on peripherals and interfaces
  - New section on using peripherals in your applications
  - New section on Asyncio/Interrupts
  - New section on trace buffer

Version 1.3

Image release: pynq_z1_image_2016_09_14

Documentation updated: 16 Dec 2016

- **Added new iop modules to docs**
  - Arduino Grove Color
  - Arduino Grove DLight

  - Arduino Grove Ear HR
  - Arduino Grove Finger HR
  - Arduino Grove Haptic motor
  - Arduino Grove TH02
  - Pmod Color
  - Pmod DLight
  - Pmod Ear HR
  - Pmod Finger HR
  - Pmod Haptic motor
  - Pmod TH02
- Added USB WiFI driver

CHAPTER 25

Indices and tables

- genindex
- modindex
- search

p

Symbols

A

B

C

D

E

F

G

R

S

T

U

V

W

X