PYNQ™

Overlay Design Methodology

XILINX®

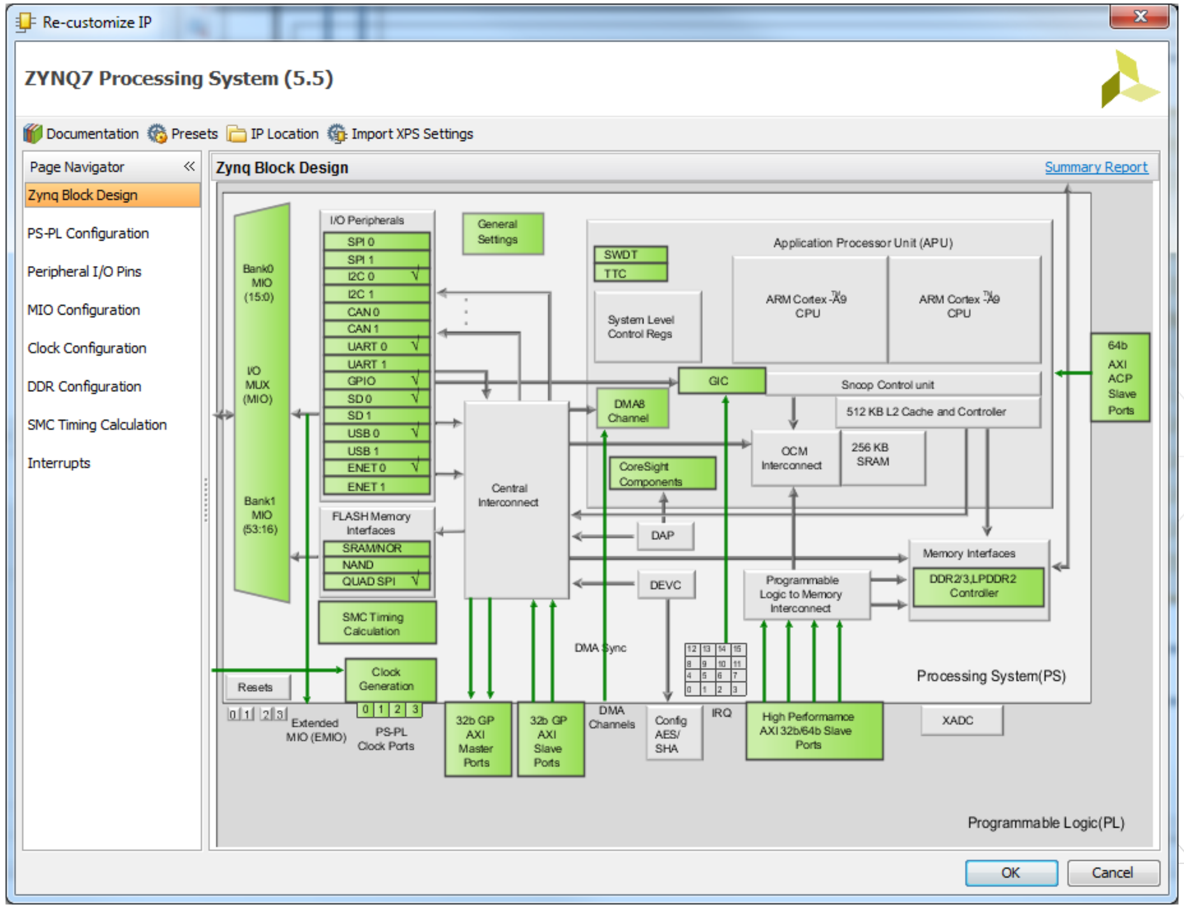
Overlay design considerations

Overlay design considerations

- Zynq design
  - Zynq PS configuration settings
  - Zynq architecture & interfaces between PS-PL
  - PL design – reusable IP
  - Overlay programmability
- Python API
  - C integration
- Soft processors
  - Software design for any IOPs or other soft processor
- Python Packaging

Zynq base configuration

- **PYNQ image used to boot board**
  - PS is configured during boot
  - Default overlay loaded
  - Some PS settings can be changed at runtime
    - E.g. PS to PL clocks
- **PS settings**
  - SD, Ethernet, USB, UART, DDR3 (Used by PYNQ)
  - Used in some overlays: Flash, GPIO (Interrupts), IIC (Video)
  - Interrupts enabled

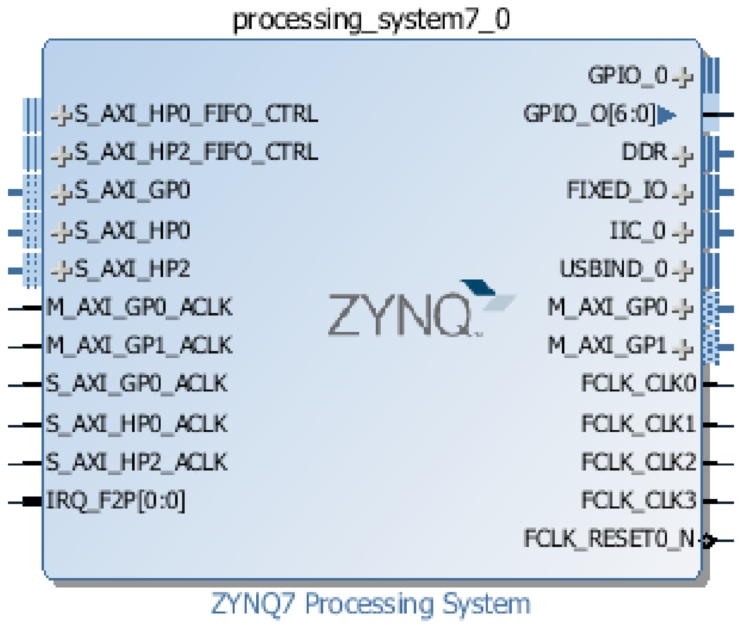


Zynq PS settings

Designing and using overlays

> **Zynq design**
>> Retain existing "PYNQ" MIO peripherals
– SD, Ethernet, USB, UART, DDR3
– May be possible to extend with EMIO
>> Use any of available PS-PL interfaces
– Do not need to be enabled in default (boot) overlay
>> Modify PS to PL clocks as required

> **Downloading new overlays**
>> The PL design (bitstream) will be downloaded by default when overlay is instantiated
>> PS clock settings to be updated from overlay Tcl/HWH



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

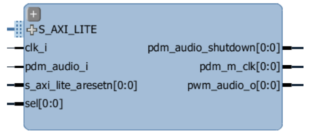
Zynq PS settings

PL Design

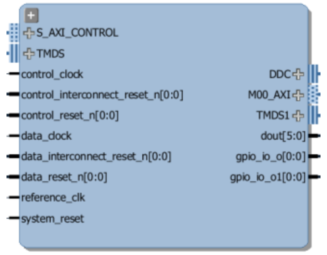
> **Standard FPGA/Zynq design**

> **V2.4: PYNQ supports Vivado: 2018.2**
  - IP versions and drivers depend on Vivado version
    - PYNQ drivers also depend on version
  - Other versions may work, but are untested (IP usually have minor revisions between versions)

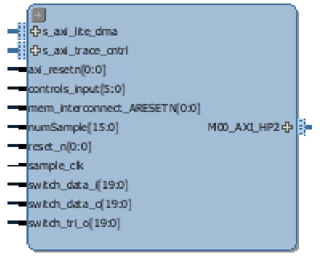
> **PYNQ reusable blocks**
  - Audio, Video, Tracebuffer, IOPs



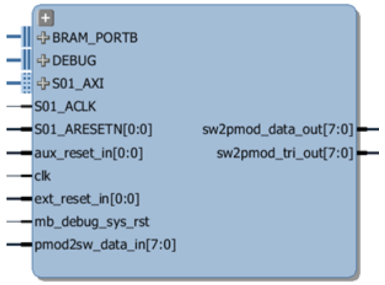
Audio



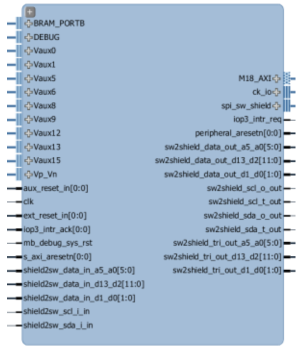
Video



Tracebuffer



Pmod IOP



Arduino IOP

Overlay Programmability

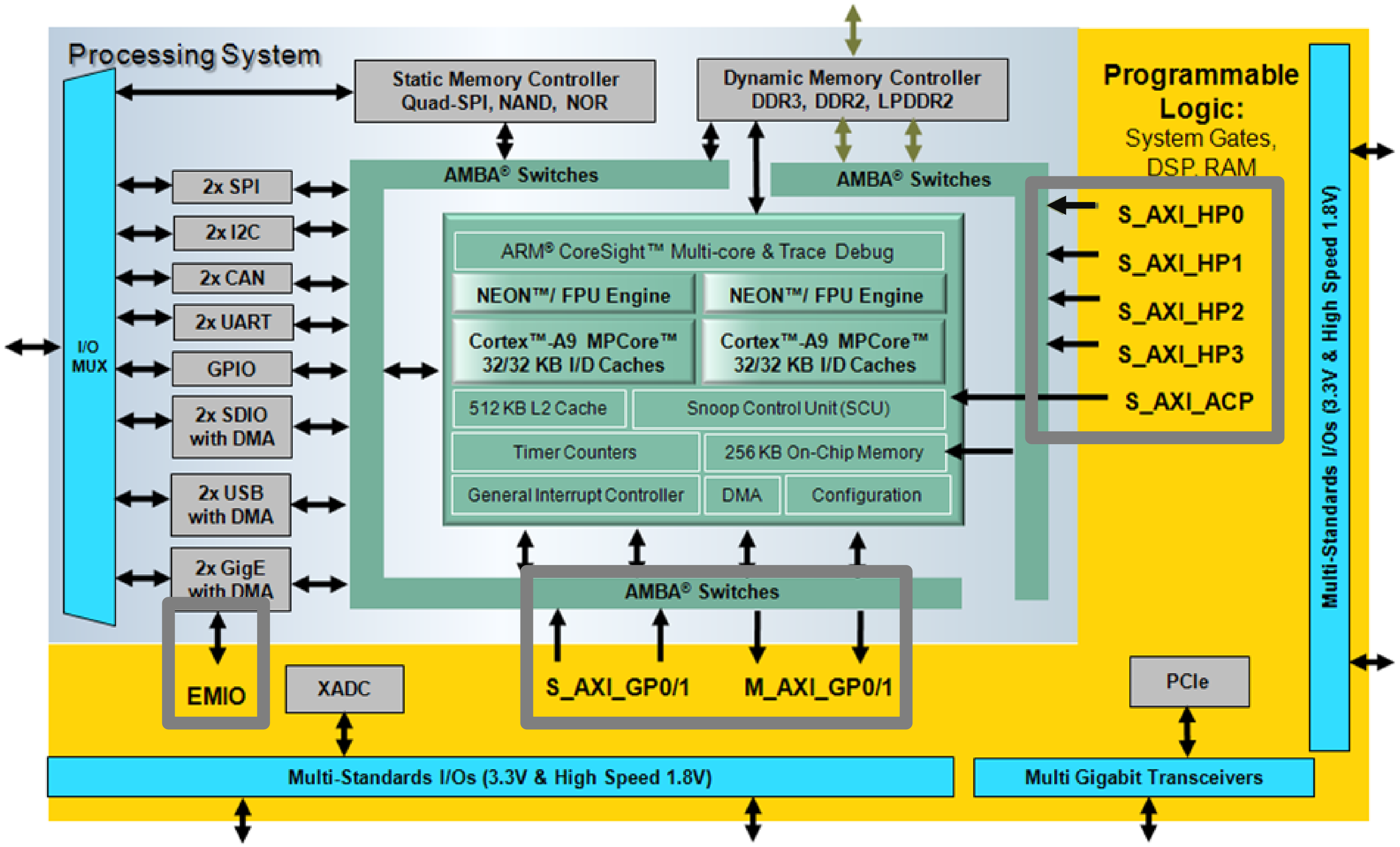
- **End-user programmability and flexibility valued over logic utilisation**
  - Create Programmable IP and overlays that can be reused for many applications and domains
- **IOPs support many peripherals from the same overlay design**
- **Performance vs flexibility**
  - For example; Neural Network Design
    - Weights can be "hard coded" into bitstream for higher performance/resource utilisation
    - However, each application may have a specific network that would need to be rebuilt
  - Neural Network Overlay
    - Weights can be programmed into BRAM at runtime for flexibility
    - Many networks can run on the same overlay

Overlay Tcl (moving to HWH)

- **Bitstream + Tcl or HWH must be provided to load an overlay**
  - Default location: \\pynq\xilinx\pynq\overlays\<overlay>\
- **Must generate Tcl from Vivado IP Integrator Block Diagram**
  - “Standardizes” Tcl scripts
  - File > Export > Export Block Design
  - write_bd_tcl command
- **Tcl/HWH and bitstream file names should match**
- **Tcl/HWH is used to provide information about the system**
  - List of IP in the design
  - GPIO information, used to connect IOP resets/interrupts
  - PS to PL clock setting, used to dynamically update clocks

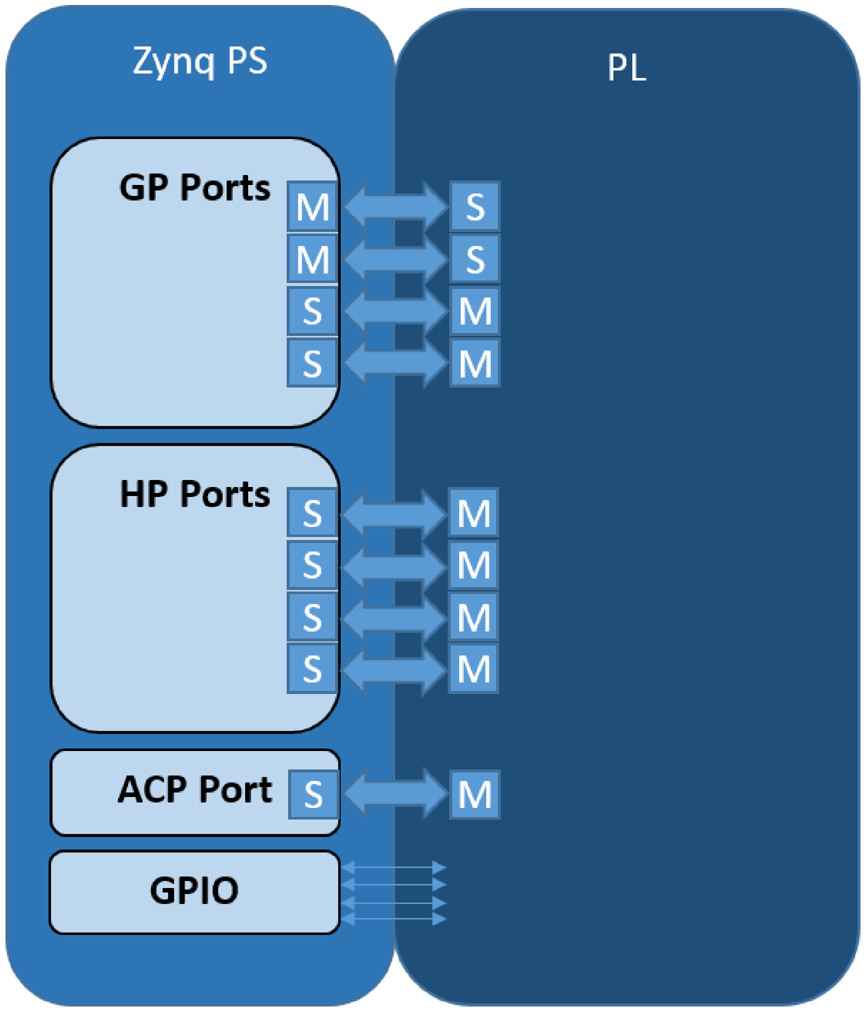
Zynq PS-PL interfaces

Zynq PS-PL interfaces



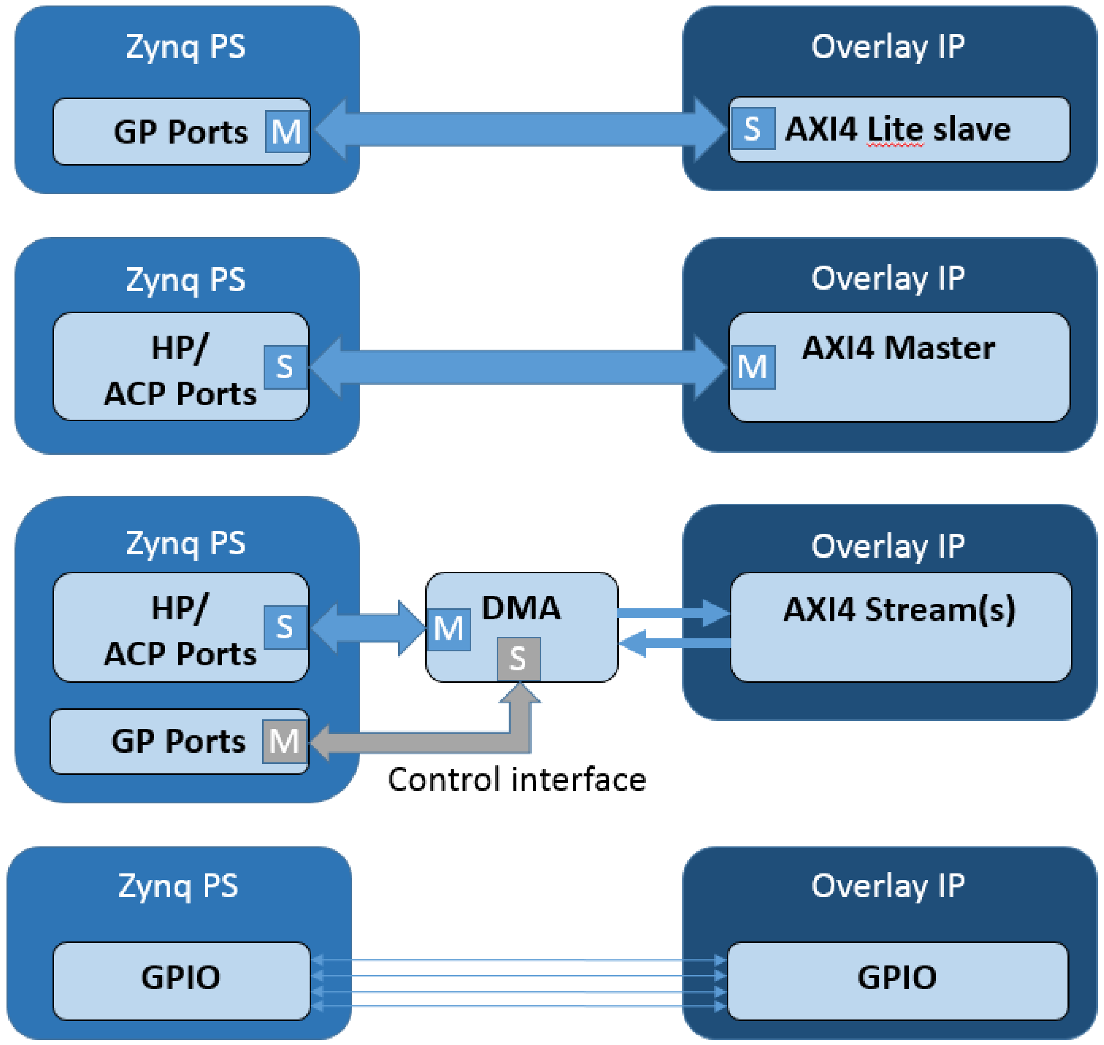
XILINX

PS-PL interfaces types



- **AXI General Purpose ports**
  - 2x Master (32-bit)
  - 2x Slave (32-bit)
- **AXI High Performance**
  - 4x Slave (64-bit)
    - 1K FIFOs
- **ACP**
  - 1x Slave (64-bit)
  - Cache access
- **GPIO (EMIO)**
  - Wires (64)

IP interfaces in PL



> **AXI (Lite) Slave IP**
>> General Purpose ports
>> Typically lower performance IP

> **AXI Master**
>> AXI HP/ACP
>> Typically higher performance IP

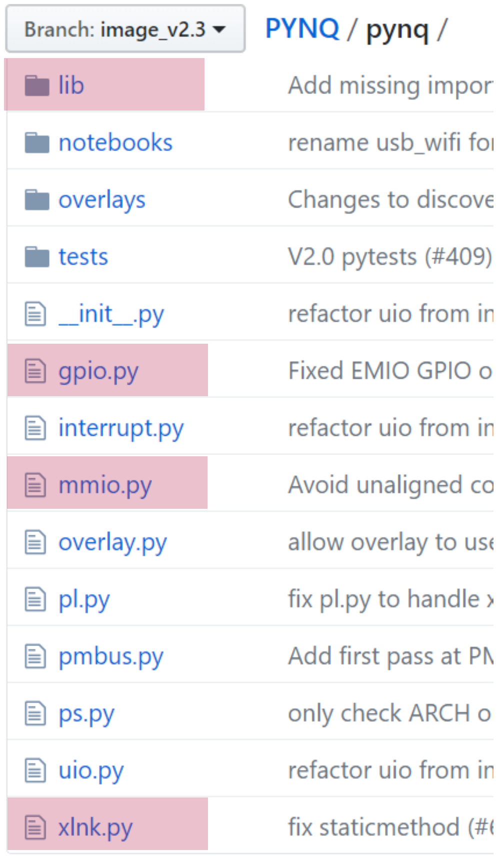
> **AXI Stream**
>> Via DMA
>> HP/ACP ports for data path
>> GP slave for control

> **GPIO**
>> Control type data

Pynq interface classes

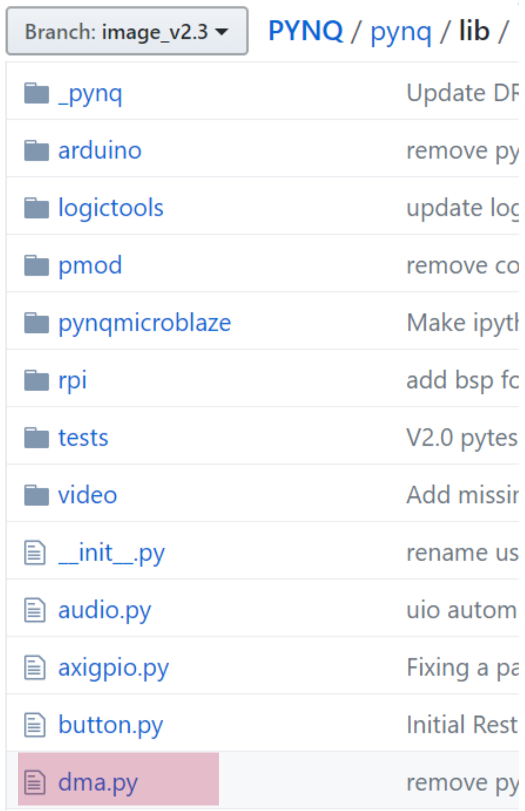
Pynq interface classes – run on PS

- **MMIO (pynq.mmio)**
  - Memory Mapped Input Output
  - Register based memory mapped transactions
- **Xlnk (pynq.xlnk)**
  - Memory allocation (used in DMA or for AXI Master peripheral)
- **DMA (pynq.lib.dma)**
  - Direct Memory Access
  - Offload memory transfers from main CPU
- **GPIO (pynq.gpio)**
  - Read/Write GPIO wires



Branch: image_v2.3 ▾ PYNQ / pynq /

lib	Add missing impor
notebooks	rename usb_wifi fo
overlays	Changes to discove
tests	V2.0 pytests (#409)
__init__.py	refactor uio from in
gpio.py	Fixed EMIO GPIO o
interrupt.py	refactor uio from in
mmio.py	Avoid unaligned co
overlay.py	allow overlay to use
pl.py	fix pl.py to handle x
pmbus.py	Add first pass at PN
ps.py	only check ARCH o
uio.py	refactor uio from in
xlnk.py	fix staticmethod (#



Branch: image_v2.3 ▾ PYNQ / pynq / lib /

_pynq	Update DF
arduino	remove py
logictools	update log
pmod	remove co
pynqmicroblaze	Make ipyth
rpi	add bsp fc
tests	V2.0 pytes
video	Add missir
__init__.py	rename us
audio.py	uio autom
axigpio.py	Fixing a pa
button.py	Initial Rest
dma.py	remove py

MMIO Class – memory mapped IO

- **Import MMIO**
- **Define memory mapped region**
  - BASE_ADDRESS: starting location
  - ARRAY_SIZE: length of accessible memory (optional, default 4 bytes)
- **Read and Write 32-bit values**
  - ADDRESS_OFFSET: Offset from BASE_ADDRESS, should be multiples of 4
  - Need to ensure the memory can be read/written

```
from pynq import MMIO

BASE_ADDRESS = 0x40000000
ARRAY_SIZE = 1024

mmio = MMIO(BASE_ADDRESS, ARRAY_SIZE)

data = 0x12345678
ADDRESS_OFFSET = 0x10

mmio.write(ADDRESS_OFFSET, data)

result = mmio.read(ADDRESS_OFFSET)

print(hex(result))
> 0x12345678
```

Xlnk Class

- **Memory managed by Linux**
  - Virtual
- **Memory must be allocated before PL Master can access it**
  - PL needs the physical address of a memory buffer
- **Xlnk can allocate (contiguous) memory buffers (using NumPy)**
  - Maps virtual and physical addresses
  - Contiguous memory is more efficient/allows simpler DMA logic
  - Array can be specified as NumPy data type, and size/shape
- **Once buffer is allocated a DMA can be used to transfer data between PS/PL**
- **DMA class uses Xlnk for memory allocation**

Xlnk example

- Import Xlnk
- Allocate contiguous buffer
  - cma_array()
  - Returns Linux virtual address
- Get Physical Address
  - `.physical_address`
  - Can be used by PL to access DDR (Linux) memory
- Read/write buffer

```
MEMORY_SIZE = 10
from pynq import Xlnk
import numpy as np
xlnk = Xlnk()

input_buffer = xlnk.cma_array(shape=(10,),
dtype=np.float32)
phy_addr = input_buffer.physical_address

for i in range(10):
    input_buffer[i] = i
print(input_buffer)
```

DMA Class

- **Direct memory access**
  - Transfer data between memories directly
    - PS - PL
  - Bypasses CPU
    - Doesn’t waste CPU cycles on data transfer
  - Speed up memory transfers with burst transactions
- **Xilinx *AXI Direct Memory Access* IP block supported in PYNQ**
  - Read and Write Paths from PL to DDR and DDR to PL
  - Memory Mapped to Stream
  - Stream to Memory Mapped
- **Needs to stream to/from an allocated memory buffer**
  - PYNQ DMA class inherits from xlnk for memory allocation

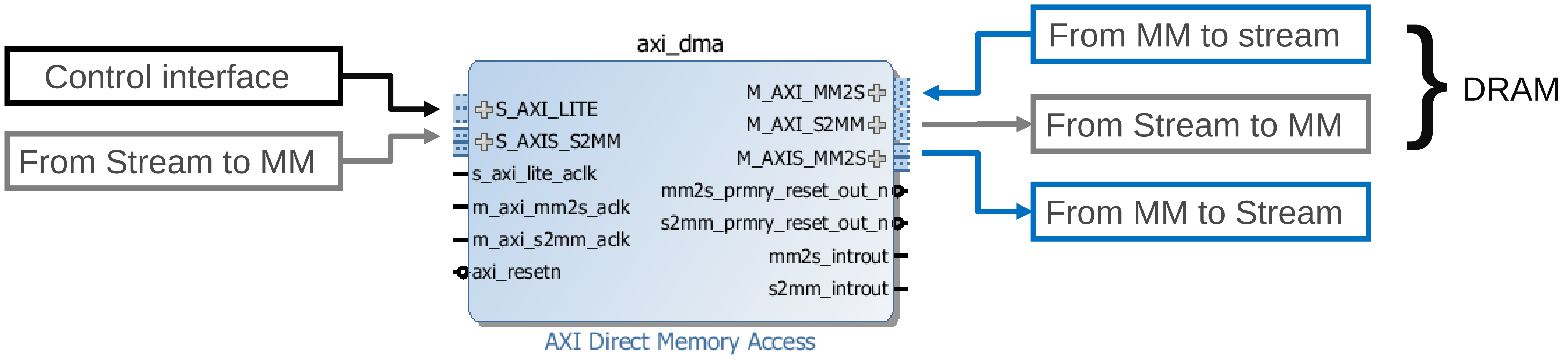
AXI Direct Memory Access IP

> **AXI Lite control interface (AXI GP port)**

> **Memory mapped interface (AXI interface, HP/ACP ports)**

> **AXI Stream interface (AXI stream accelerator)**

> **Transfer between streams and memory mapped locations**
>> Paths from PL to DRAM and DRAM to PL
>> Memory Mapped to Stream (MM2S)/Stream to Memory Mapped (S2MM)



XILINX

DMA example

- > **Setup DMAs**
- > **Allocate memory buffers**
- > **Start DMA transactions**

Create DMA instances

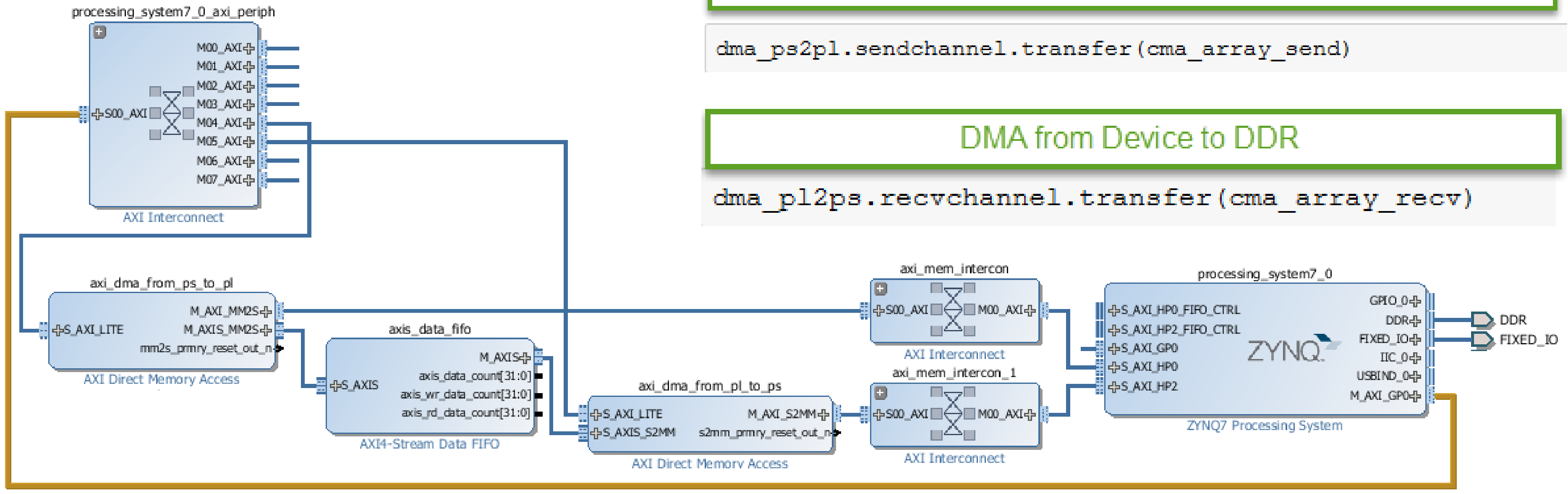
```
from pynq.lib import DMA

dma_ps2pl_description = overlay.ip_dict['axi_dma_from_ps_to_pl']
dma_ps2pl = DMA(dma_ps2pl_description)

dma_pl2ps_description = overlay.ip_dict['axi_dma_from_pl_to_ps']
dma_pl2ps = DMA(dma_pl2ps_description)
```

DMA from DDR to Device

```
dma_ps2pl.sendchannel.transfer(cma_array_send)
```

DMA from Device to DDR

```
dma_pl2ps.recvchannel.transfer(cma_array_recv)
```



XILINX

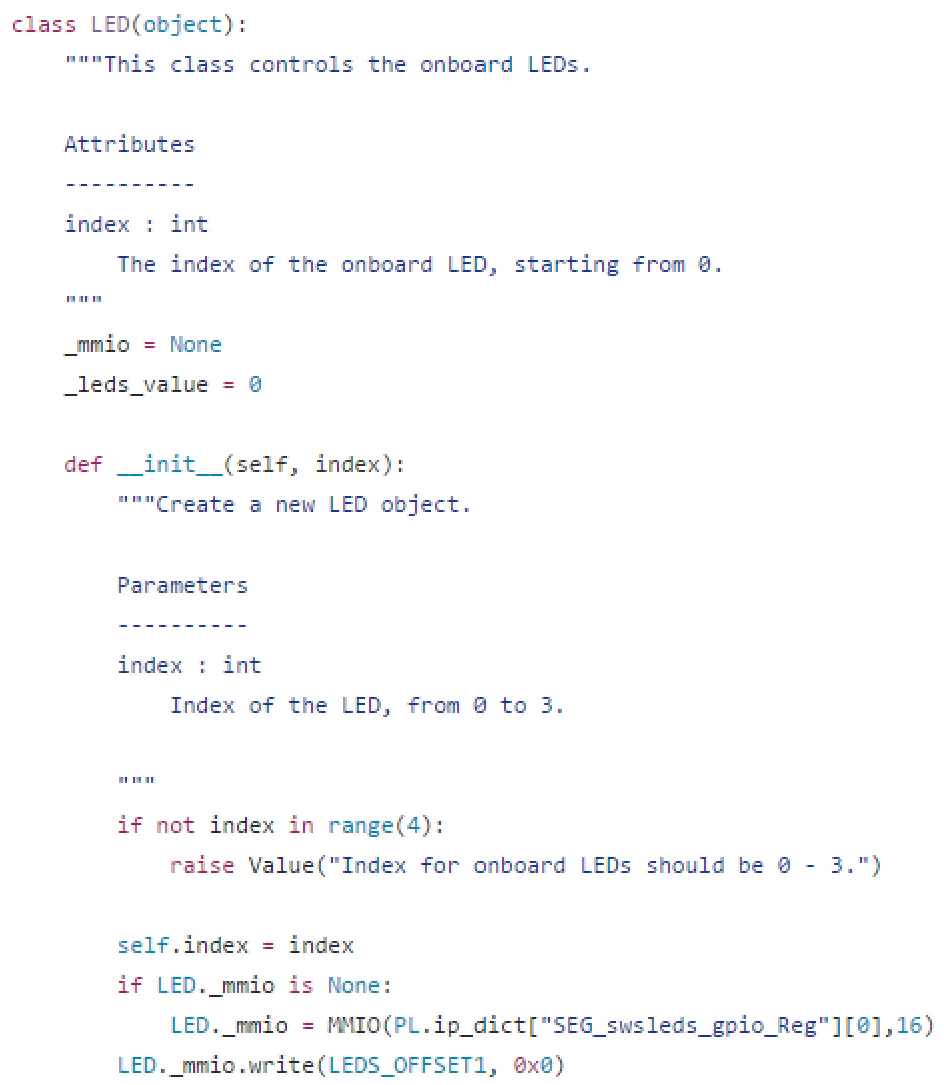
GPIO Class

- **Up to 64 GPIO wires from PS**
  - Tri-state
- **Accessed from Linux**
- **Setup GPIO instance**
- **Map to GPIO pin**
  - Translated pin numbers to Linux GPIO number
  - get_gpio_pin()
- **Read/Write**
- **Most appropriate for simple control**
  - Set, reset, start, done …

```
from pynq import GPIO

btn_gpio = GPIO.get_gpio_pin(0)
led_gpio = GPIO.get_gpio_pin(1)

ps_btn = GPIO(btn_gpio, 'in')
ps_led = GPIO(led_gpio, 'out')

ps_btn.read()

ps_led.write()
```

Python Wrapper

- > **Standard Python code**
- > **Interface classes used**
  - >> MMIO, Xlnk, DMA, GPIO
- > **Check PYNQ repository for examples**



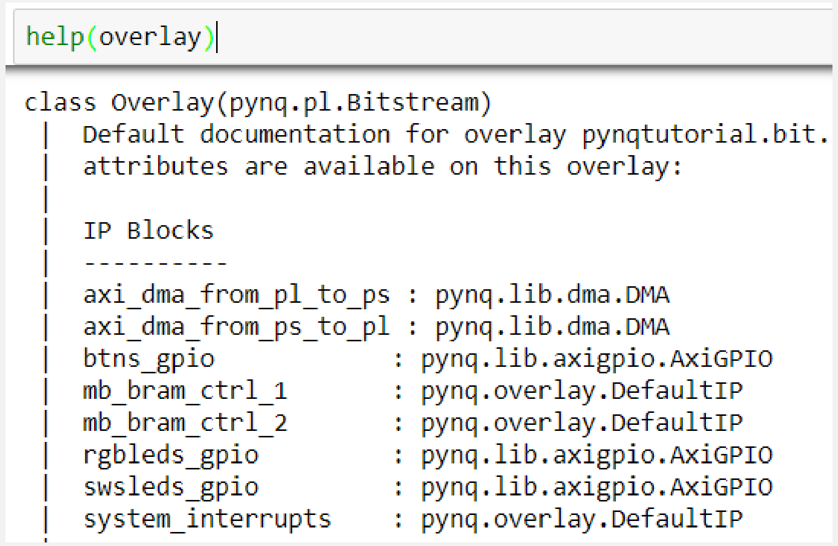
```
class LED(object):
    """This class controls the onboard LEDs.

    Attributes
    ----------
    index : int
        The index of the onboard LED, starting from 0.
    """
    _mmio = None
    _leds_value = 0

    def __init__(self, index):
        """Create a new LED object.

        Parameters
        ----------
        index : int
            Index of the LED, from 0 to 3.

        """
        if not index in range(4):
            raise Value("Index for onboard LEDs should be 0 - 3.")

        self.index = index
        if LED._mmio is None:
            LED._mmio = MMIO(PL.ip_dict["SEG_swsleds_gpio_Reg"][0],16)
        LED._mmio.write(LEDS_OFFSET1, 0x0)
```

Overlay API

Python overlay API

> **PYNQ Overlay class supports basic overlay functionality**
  - Requires Tcl/HWH
  - Allows download, and overlay discovery (ip_dict)
  - Assigns default drivers to IP
    - Read() and write() access to IP address space

```
from pynq import Overlay
overlay = Overlay("pynqtutorial.bit")
```



```
help(overlay)
```

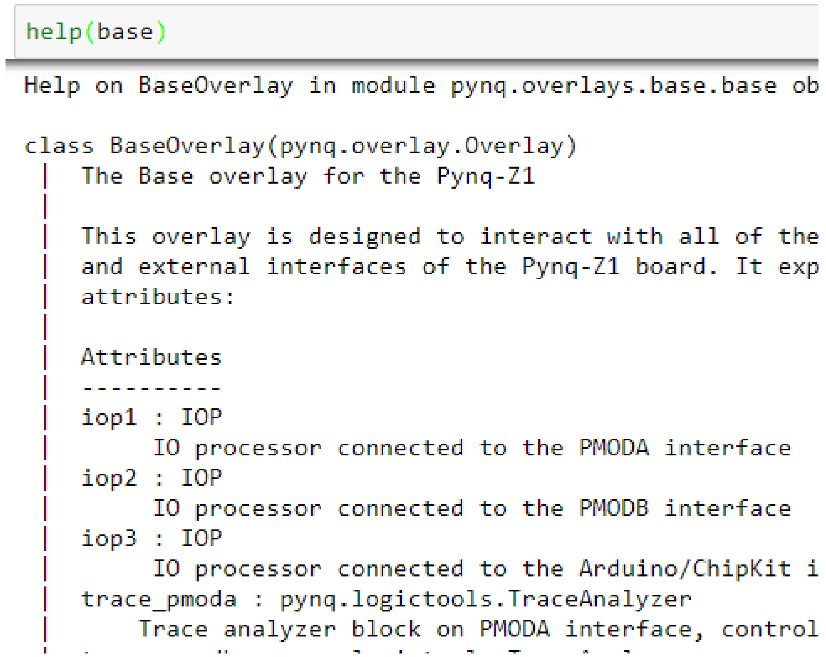
```
class Overlay(pynq.pl.Bitstream)
 |  Default documentation for overlay pynqtutorial.bit.
 |  attributes are available on this overlay:
 |
 |  IP Blocks
 |  ----------
 |  axi_dma_from_pl_to_ps : pynq.lib.dma.DMA
 |  axi_dma_from_ps_to_pl : pynq.lib.dma.DMA
 |  btns_gpio            : pynq.lib.axigpio.AxiGPIO
 |  mb_bram_ctrl_1       : pynq.overlay.DefaultIP
 |  mb_bram_ctrl_2       : pynq.overlay.DefaultIP
 |  rgbleds_gpio         : pynq.lib.axigpio.AxiGPIO
 |  swsleds_gpio         : pynq.lib.axigpio.AxiGPIO
 |  system_interrupts    : pynq.overlay.DefaultIP
```

```
overlay.rgbleds_gpio.write(0, 3)
overlay.swsleds_gpio.read()
```

Custom Overlay API

- **Custom Overlay API**
  - Allows automatic assignment of custom drivers
    - Overwrite default drivers
  - No need to import additional drivers
  - Easier for software developers to use overlays
- **Overlay “attributes” available from overlay instance**
  - E.g. base.PMODA
- **help()**
  - Discover information about the overlay

```
from pynq.overlays.base import
BaseOverlay

base = BaseOverlay("base.bit")
```



```
help(base)
```

```
Help on BaseOverlay in module pynq.overlays.base.base ob

class BaseOverlay(pynq.overlay.Overlay)
 |  The Base overlay for the Pynq-Z1
 |
 |  This overlay is designed to interact with all of the
 |  and external interfaces of the Pynq-Z1 board. It exp
 |  attributes:
 |
 |  Attributes
 |  ----------
 |  iop1 : IOP
 |       IO processor connected to the PMODA interface
 |  iop2 : IOP
 |       IO processor connected to the PMODB interface
 |  iop3 : IOP
 |       IO processor connected to the Arduino/ChipKit i
 |  trace_pmoda : pynq.logictools.TraceAnalyzer
 |      Trace analyzer block on PMODA interface, control
```

Python overlay API example: base overlay

- Custom Python wrapper provided with overlay
- Inherits from pynq.Overlay class
- Define custom drivers, interface types, IOP types, etc.
- If no driver is specified, a *DefaultIP* driver is assigned
  - Read(), write() to IP address space (base on MMIO)

```
class BaseOverlay(pynq.Overlay):

        self.iop1.mbtype = "Pmod"
        self.iop3.mbtype = "Arduino”

        self.leds =
                  self.swsleds_gpio.channel2
        self.switches =
                  self.swsleds_gpio.channel1

        self.setlength(4)
        self.switches.setlength(2)

        self.leds.setdirection("out")
        self.switches.setdirection("in")
```

xilinx\pynq\overlays\base\base.py

Pynq Python package

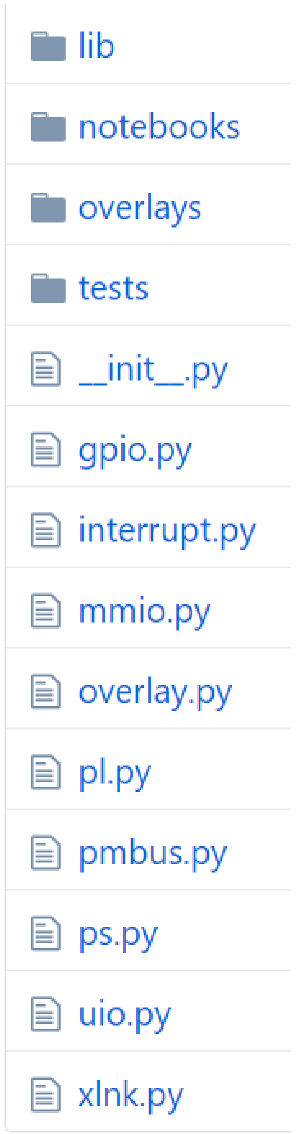
PYNQ Python package

> **pynq/**
  - Interface classes
  - Overlays
  - Tests

> **pynq/lib**
  - Pmod, Grove, Arduino, RPi driver classes
  - *logictools* class
  - Video, Audio, DMA, Tracebuffer, WiFi

> **pynq/overlays**
  - Application domains

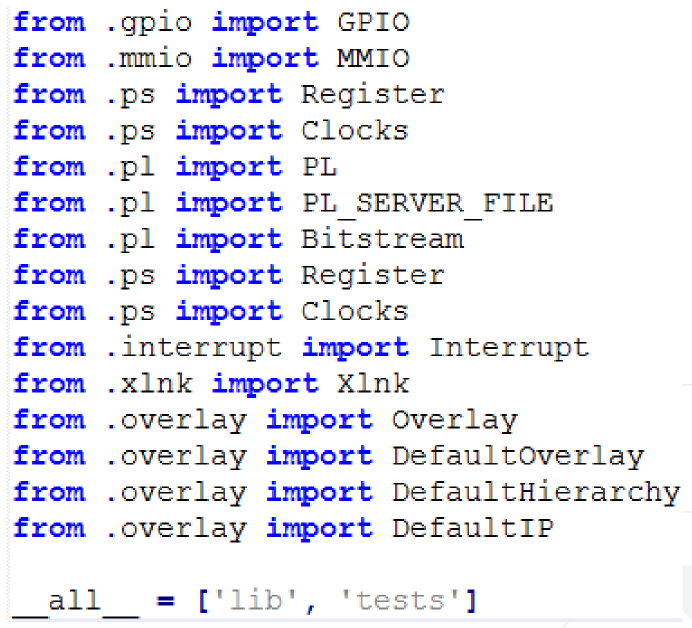
> **pynq/tests**
  - Pytests for basic modules



lib
notebooks
overlays
tests
__init__.py
gpio.py
interrupt.py
mmio.py
overlay.py
pl.py
pmbus.py
ps.py
uio.py
xlnk.py

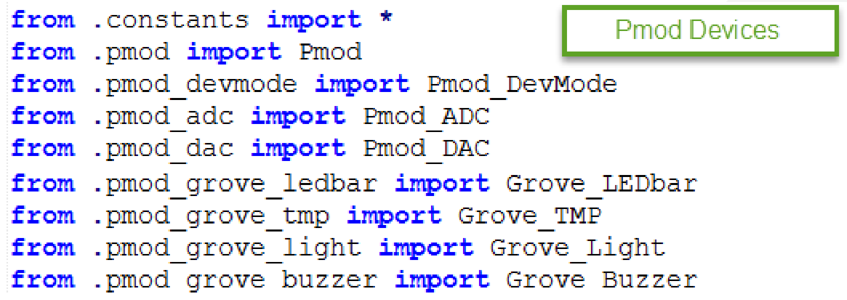
Add new Python modules to PYNQ

- **__init__.py**
  - Exists in each directory
  - Include filename and class to be imported
- **Hierarchal**
  - Top level can include lower level directory



```python
from .gpio import GPIO
from .mmio import MMIO
from .ps import Register
from .ps import Clocks
from .pl import PL
from .pl import PL_SERVER_FILE
from .pl import Bitstream
from .ps import Register
from .ps import Clocks
from .interrupt import Interrupt
from .xlnk import Xlnk
from .overlay import Overlay
from .overlay import DefaultOverlay
from .overlay import DefaultHierarchy
from .overlay import DefaultIP

__all__ = ['lib', 'tests']
```

pynq\__init__.py



```python
from .constants import *
from .pmod import Pmod
from .pmod_devmode import Pmod_DevMode
from .pmod_adc import Pmod_ADC
from .pmod_dac import Pmod_DAC
from .pmod_grove_ledbar import Grove_LEDbar
from .pmod_grove_tmp import Grove_TMP
from .pmod_grove_light import Grove_Light
from .pmod_grove_buzzer import Grove_Buzzer
```

pynq\lib\pmod\__init__.py

Packaging and distributing overlays

Distributing Overlays

- **"Overlay" (optionally) consists of**
  - Bitstream (.bit)
  - Tcl (.tcl)/HWH (.hwh)
  - Python (.py)
  - Notebook (.ipynb)
  - Documentation
  - C library (.so), source files, header files…
- **Pip package management can be used to install PYNQ packaged**
  - `pip install` (pip3.6 install in current image)
  - Include setup.py for pip

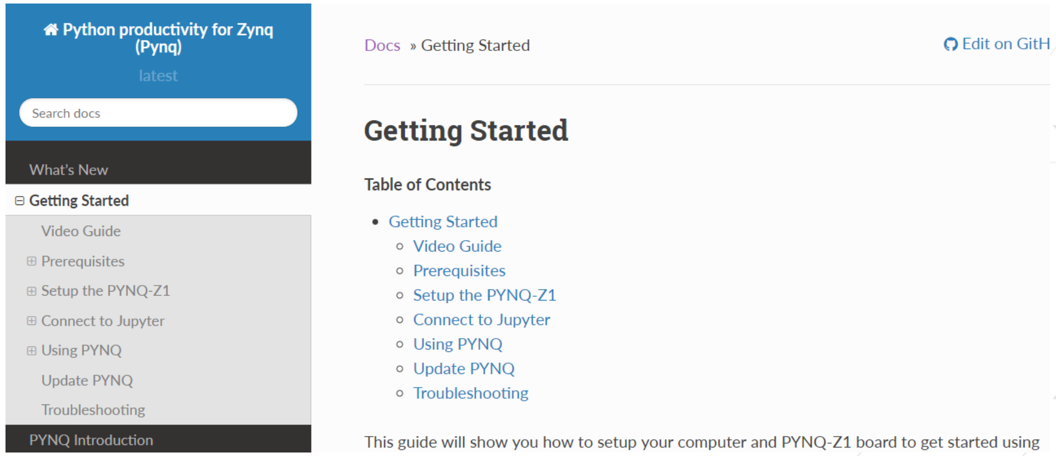
Packaging and distribution guidelines

- **Distribute standalone Python package**
  - Not a fork of PYNQ
- **GitHub recommended**
  - Include readme.md - displays on GitHub home page
  - Bug tracking, feature request, version management
  - Jupyter Notebooks render on GitHub – easy to share and demonstrate work
- **Specify License**
  - BSD3 preferred (compatible with other open source licenses)
- **Verification**
  - py.test for regression should ship with package
- **Pip installable**

https://jeffknupp.com/blog/2013/08/16/open-sourcing-a-python-project-the-right-way/

Example package directory structure

```
|- LICENSE
|- README.md
|- setup.py
|- project
|    |-- __init__.py
|    |-- *.py
|    |-- test
|         |-- *.py
|         |-- test_project.py
|- docs
|    |-- conf.py
|    |-- index.rst
|    |-- *.rst
|- requirements.txt
|- TODO.md
```



Python productivity for Zynq (Pynq)
latest
Search docs

- What's New
- Getting Started
  - Video Guide
  - Prerequisites
  - Setup the PYNQ-Z1
  - Connect to Jupyter
  - Using PYNQ
  - Update PYNQ
  - Troubleshooting
- PYNQ Introduction

Docs » Getting Started   Edit on GitH

Getting Started

Table of Contents

- Getting Started
  - Video Guide
  - Prerequisites
  - Setup the PYNQ-Z1
  - Connect to Jupyter
  - Using PYNQ
  - Update PYNQ
  - Troubleshooting

This guide will show you how to setup your computer and PYNQ-Z1 board to get started using

- **Read The Docs**
  - Automatically generate documentation from GitHub repository
- **Include**
  - conf.py – configuration file
  - *.rst (Markdown files)
  - Jupyter notebooks

Pip install example setup.py (BNN)

- **setuptools**
  - Used by pip
- **Package data references all files to be installed**
  - Will be delivered to package directory
- **data_files can be used to install outside the package directory**

```
from setuptools import setup, find_packages
import pynq_proj

setup(
    name = "pynq_proj",
    version = pynq_proj.__version__,
    url = 'https://github.com/pynq_proj',
    license = 'Apache Software License',
    author = "Cathal McCabe",
    author_email = "cmccabe@xilinx.com",
    packages = ['ws2812'],
    package_data = {
    '' : ['*.bit','*.tcl','*. py'],
    },
    description = "PYNQ Project ..."
)
```

https://github.com/Xilinx/BNN-PYNQ/blob/master/setup.py

Summary

> **Overlay Design considerations**
>> PL, PS configuration, Programmability, Python interface

> **Zynq PS-PL interfaces**
>> HP/GP, EMIO GPIO

> **PYNQ interface classes**
>> MMIO, Xlnk, DMA, GPIO

> **PYNQ Python package**

> **Packaging and distributing overlays**
>> Readme, License, Standalone, Pip install, Verification

Adaptable.
Intelligent.