GPIOs on USRPs: The definitive guide

Everyday I'm toggling

Martin Braun

Chief Engineer

What is a GPIO? And what's it doing on a USRP?

- Wikipedia: "A **general-purpose input/output** (**GPIO**) is an uncommitted digital signal pin on an integrated circuit or electronic circuit board which may be used as an input or output, or both, and is controllable by software. [...] GPIOs have no predefined purpose and are unused by default."
- Basically, this is what makes Raspberry Pis, Arduinos useful and attractive.
- We have those on USRPs, too!

These Things

Part 1: Light ‘em up!

Everyone starts with blinking LEDs

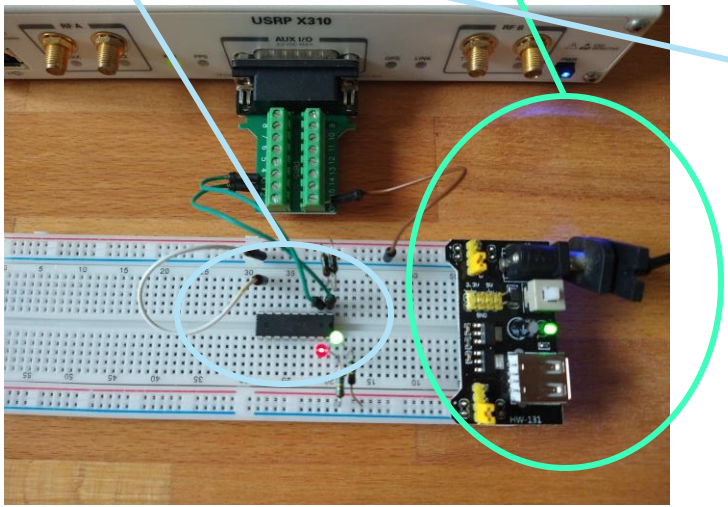
Hello World of GPIOs: Blinking an LED

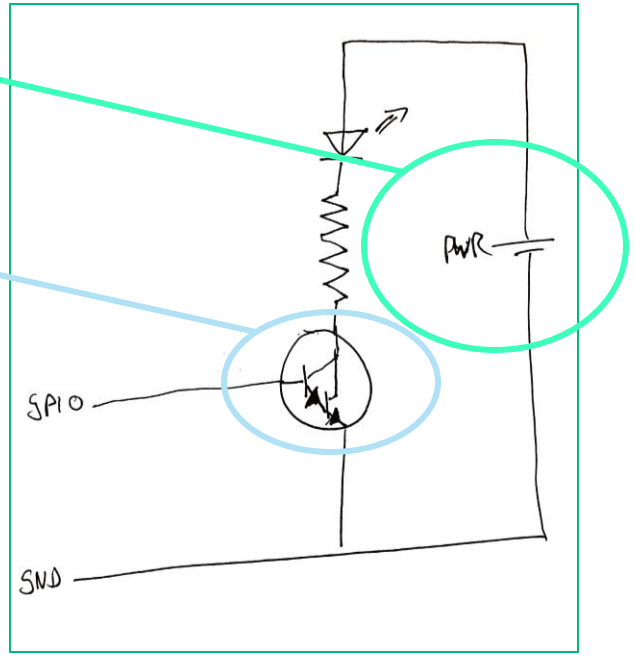
- Let's rock & roll!

Never source/sink current from GPIO pins! 5 mA max!

Hello World of GPIOs: Blinking an LED

- Use **separate power supply** to provide current (e.g., lab supply, or some USRP have power pins for this use case)
- **De-couple** GPIOs from your circuit (e.g., with Darlington)
- **Don't draw/sink** too much current (< 5 mA)!

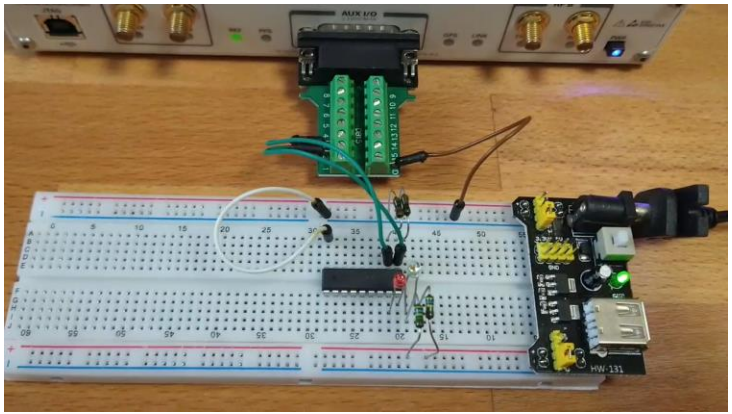




Hello World of GPIOs: Blinking an LED

- Set up GPIOs
- Toggle pin on/off

```
import uhd
import time
U = uhd.usrp.MultiUSRP("type=x300")
U.set_gpio_attr('FP0A', 'CTRL', 0x000, 0xFFF)
U.set_gpio_attr('FP0A', 'DDR',  0xFFF, 0xFFF)
for i in range(11):
    U.set_gpio_attr('FP0A', 'OUT', i % 2, 0xFFF)
    time.sleep(.5)
```

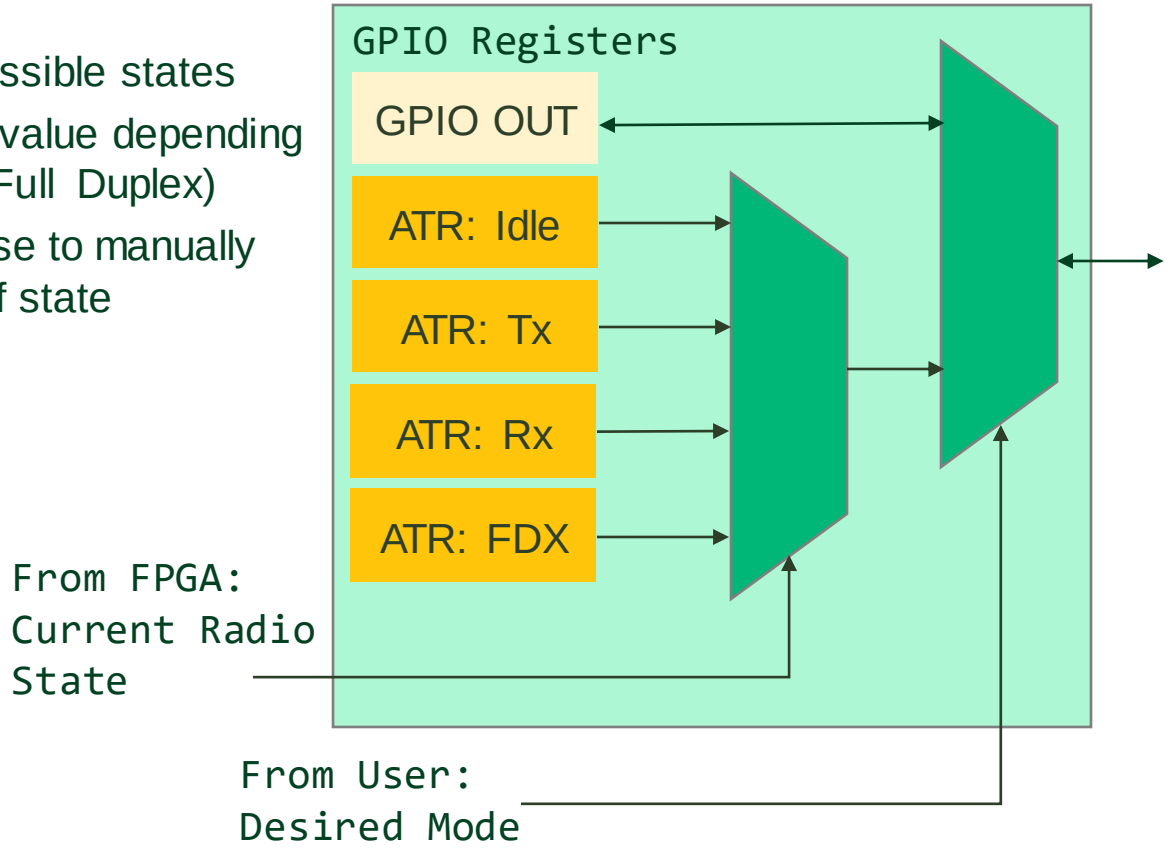


Part 2: Tracking radio state

Using ATRs

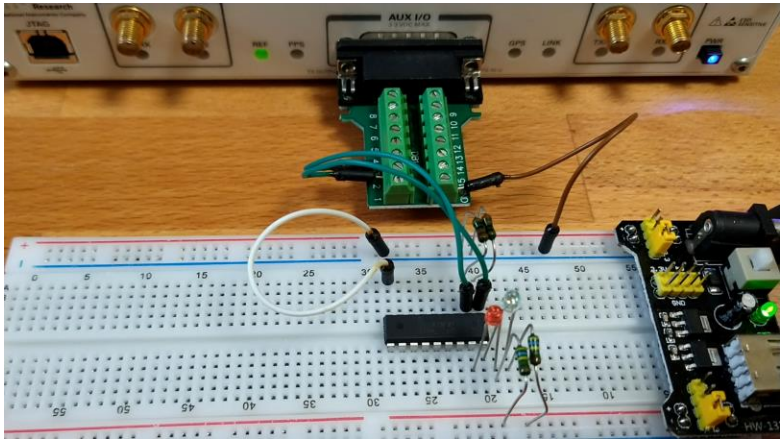
ATRs: Registers that follow TRX state

- Every radio channel has four possible states
- Every GPIO pin can be set to a value depending on the current state (Tx/Rx/Idle/Full Duplex)
- Alternatively, the user can choose to manually program the GPIO regardless of state



ATRs: Registers that follow TRX state

- How do our LEDs know when to light up green or red? They are connected to ATR registers!
- Front-panel pins can also follow ATR state, e.g., to control amplifiers
- [show animation of RX and TX, with a custom LED following the TX/RX state



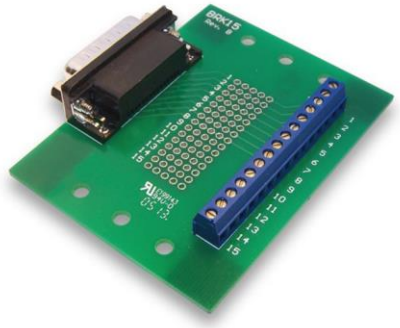

```
import time
import uhd
U = uhd.usrp.MultiUSRP("type=x300")
U.set_gpio_attr('FP0A', 'CTRL',    0x003, 0xFFF) # Pin0/1=ATR
U.set_gpio_attr('FP0A', 'DDR',     0x003, 0xFFF)
U.set_gpio_attr('FP0A', 'ATR_0X', 0x000, 0xFFF) # Idle=Off
U.set_gpio_attr('FP0A', 'ATR_RX', 0x001, 0xFFF) # RX=Pin0
U.set_gpio_attr('FP0A', 'ATR_TX', 0x002, 0xFFF) # TX=Pin1
U.set_gpio_attr('FP0A', 'ATR_XX', 0x003, 0xFFF) # FDX=Both
data = U.recv_num_samps(num_samps=int(1e6), freq=1e9, rate=1e6, gain=0)
time.sleep(1)
U.send_waveform(data, duration=1, freq=1e9, rate=1e6, gain=0)
```

Part 3: Connecting to GPIOs

Connector Diversity

X3x0 / N3x0

- DB15 Connectors (Joystick-Connector)
- GPIOs are 3.3V
- 12 pins
- Use Breakout for experimentation
- Switching speed < 10 MHz



X410

- Dual HDMI, 12 pins per connector
- Controllable GPIO voltage (1.8/2.5/3.3V)
- 3.3V Default GPIO voltage
- Dedicated 3.3V pin for power (up to 450 mA)
- Use breakout boxes for experimentation
- Switching speed < 100 MHz
- SPI-mode available

Configuring External Power Supply

The X410's GPIO ports each have 3.3V power supply pins, which is disabled by default. The GPIO lines will function correctly without the external power supply enabled, and the voltage of the power supply is independent of the selected GPIO line voltage. To enable the power supply, call the **uhd::features::gpio_power_iface::set_external_power()** method on the gpio_power discoverable feature attached to the mb_controller:

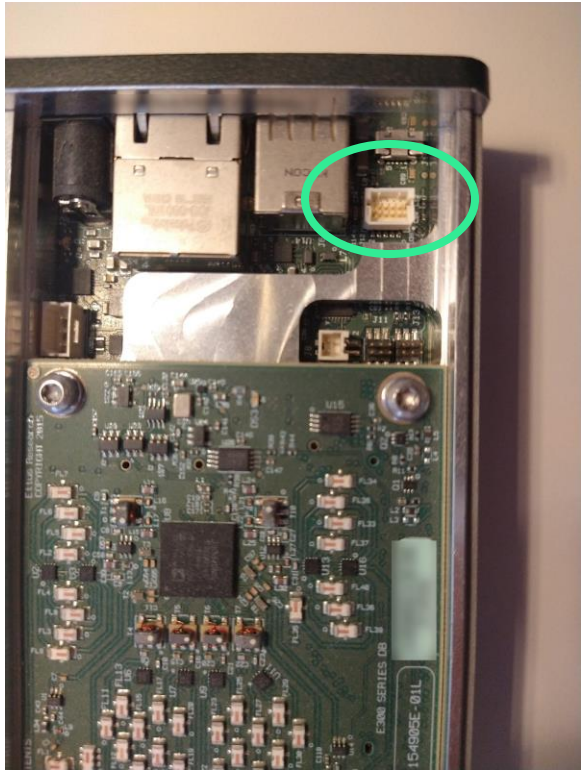
```
auto usrp = uhd::usrp::multi_usrp::make("type=x4xx");

auto& gpio = usrp->get_mb_controller().get_feature<uhd::features::gpio_power_iface>();
gpio.set_external_power("GPIO1", true); // Enable external power on GPIO1
```

E320

- Mini-HDMI Connector on back-panel
- Recommendation: Use Mini-HDMI to HDMI cable, then breakout like regular HDMI GPIO
- 8 GPIOs available
- 3.3V GPIO voltage

E310

- Internal pins only
  (requires opening enclosure)
- 6 GPIOs (3.3V)
- +3.3V pin allows drawing up to 500 mA



Internal GPIO

Connector

1	3	5	7	9
2	4	6	8	10

J12

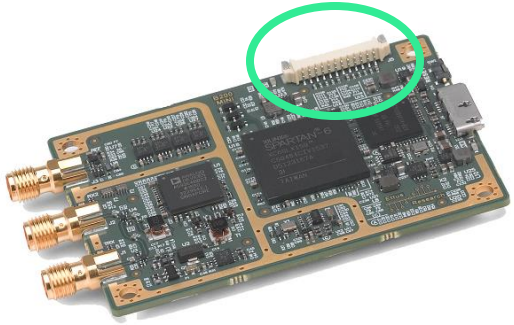
E31x GPIO Connector

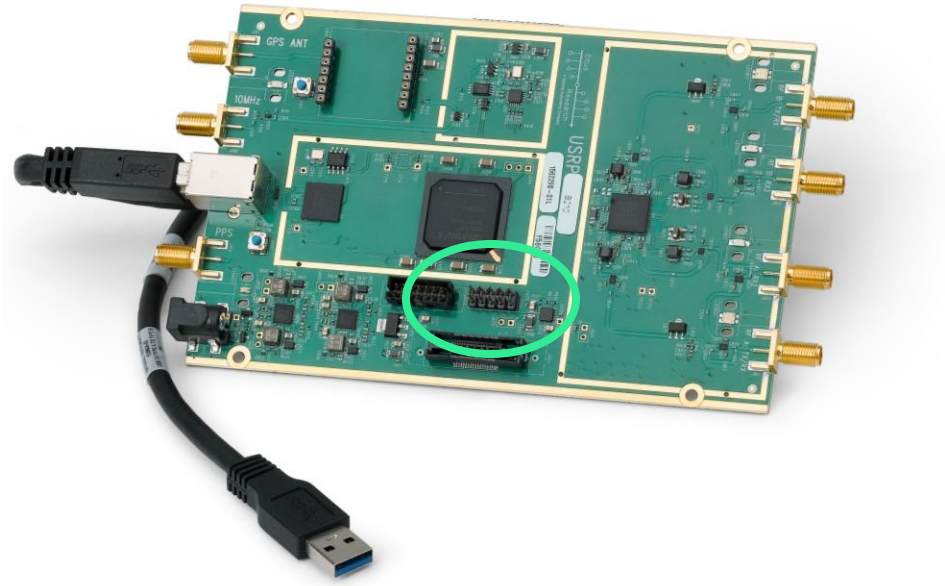
Pin Mapping

- Pin 1: +3.3V
- Pin 2: I2C SCL (3.3 V)
- Pin 3: Data[5]
- Pin 4: I2C SDA (3.3 V)
- Pin 5: Data[4]
- Pin 6: Data[0]
- Pin 7: Data[3]
- Pin 8: Data[1]
- Pin 9: 0V
- Pin 10: Data[2]

B2x0-mini and B2x0

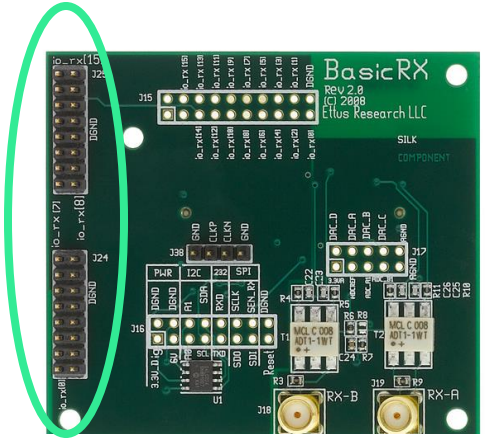
- 8 pins, 3.3V
- Mini-series: Adapter required! (Available on ettus.com)
- B200/B210: Use ribbon cable from J504 (standard 100mil spacing)
  - J504 may not be populated (depending on rev/product)

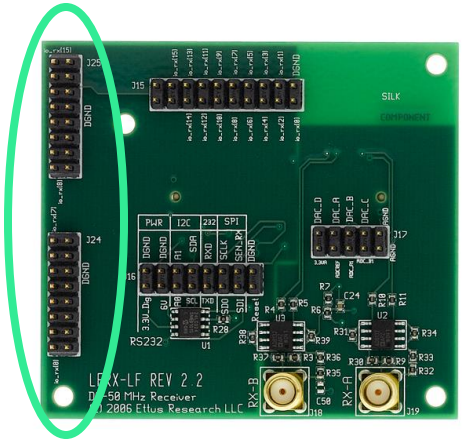


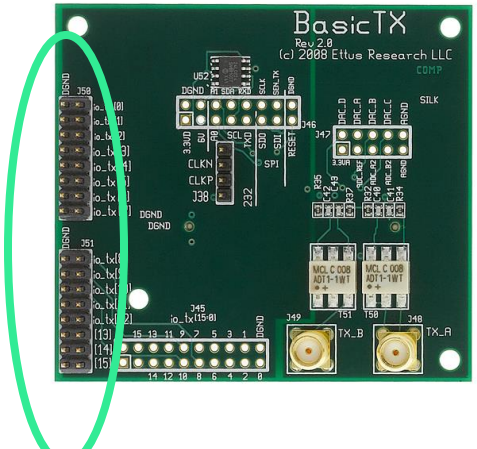


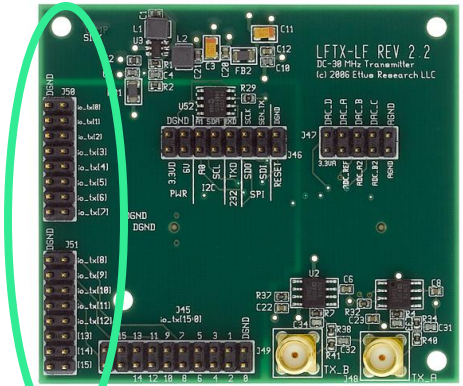
Basic/LF Daughterboards

- Only way to get GPIOs on N2x0/USRP2/USRP1
- Also works on X3x0
- Connected to TX/RX banks, not FP0 banks





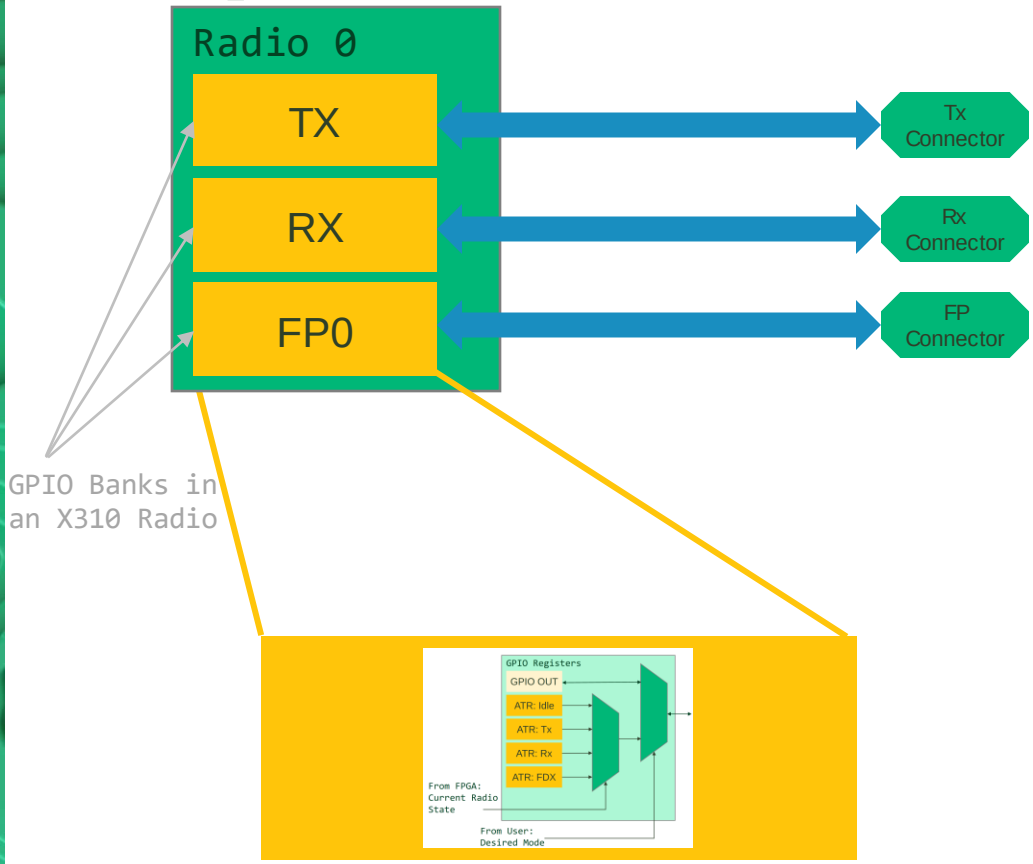




Part 4: Software Control / APIs

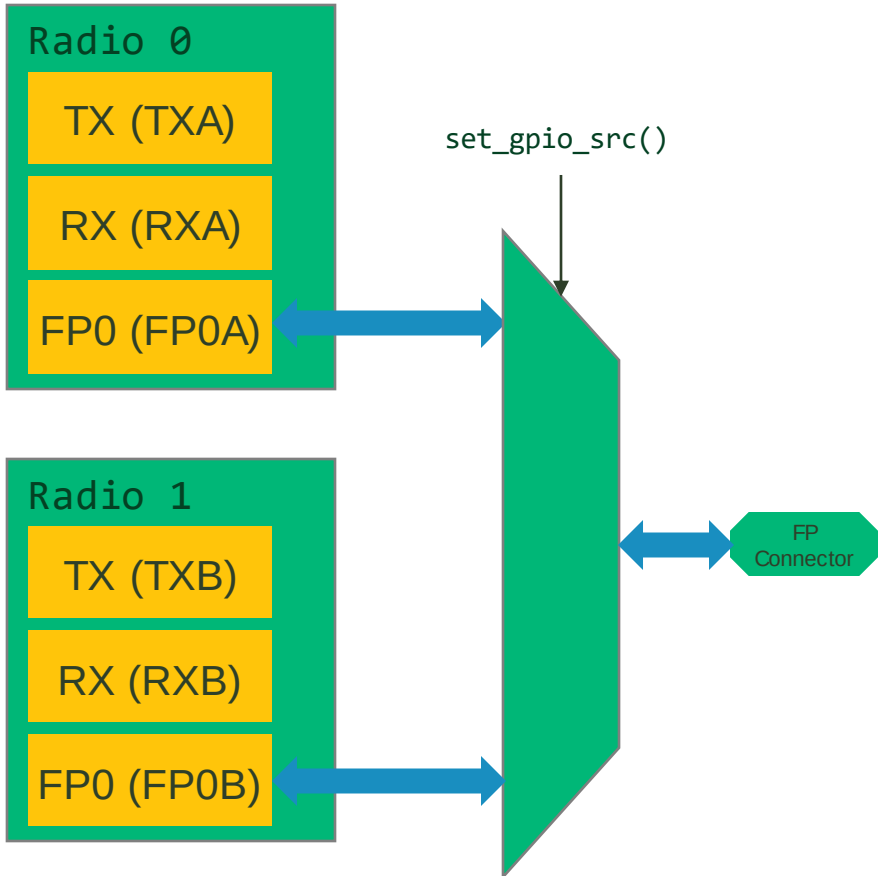
General Concepts and Software Usage

Concepts & Nomenclature: Banks



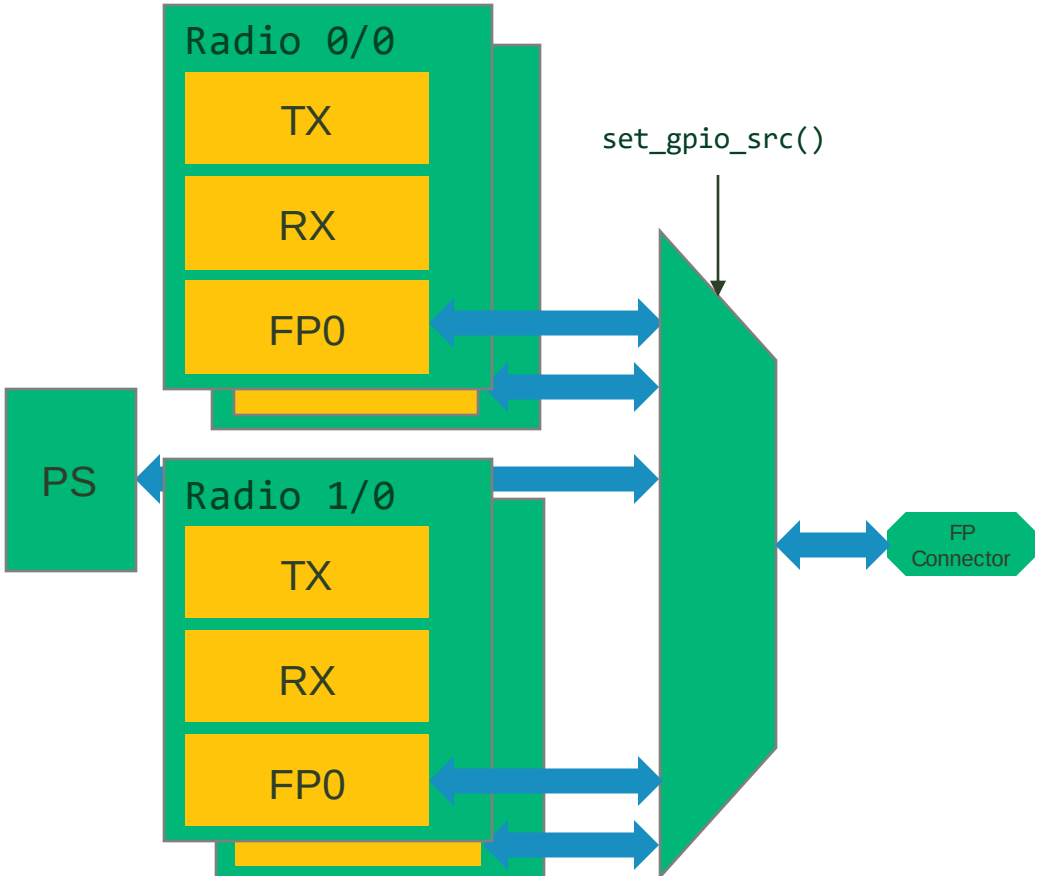
- GPIO “Banks” are SW-controllable GPIOs in the FPGA, connected to specific pieces of hardware
- UHD lets you query available banks using the get_gpio_banks() multi_usrp API
- For example, in X310, every Radio has three banks (TX, RX, FP) to control different components
  - Unless using a BasicRX/TX or LFRX/TX board, you should probably not use the RX and TX GPIO banks
- multi_usrp appends “A” and “B” to distinguish banks that have the same name on different radios

Concepts & Nomenclature: Sources



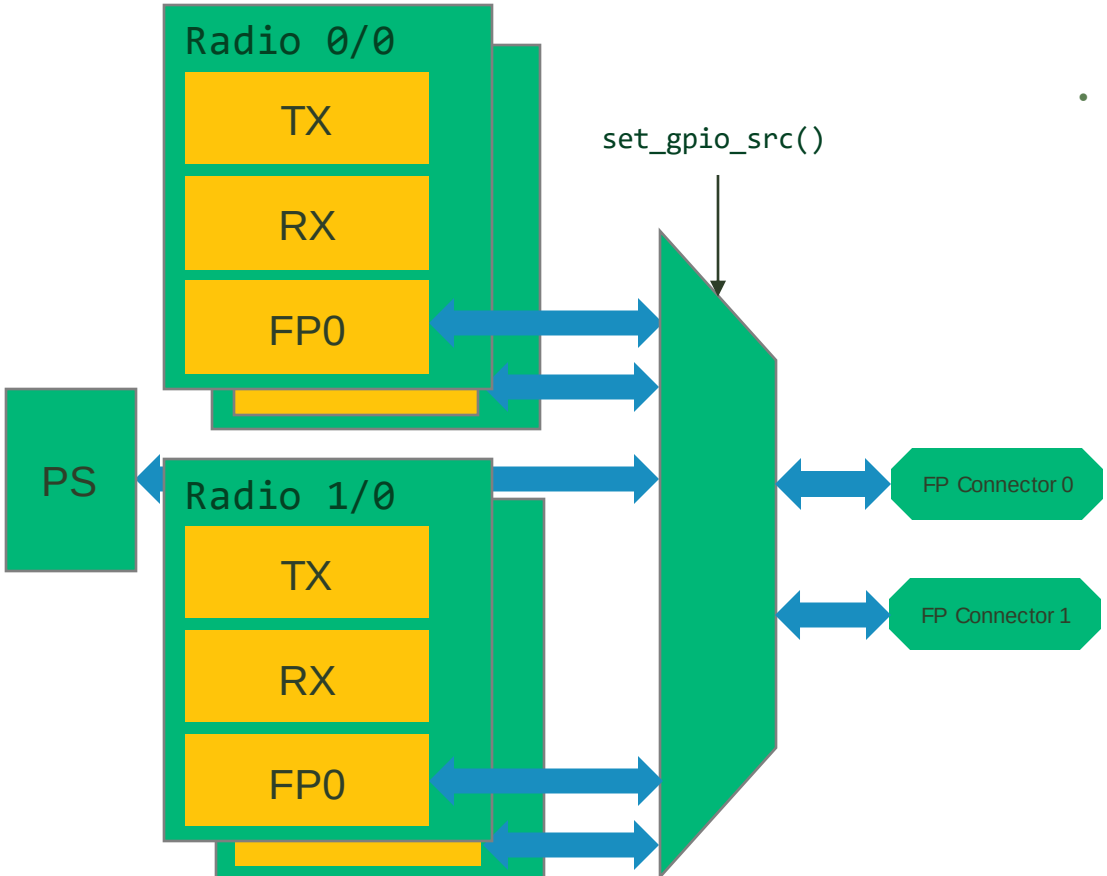
- X310 has two radios though! Which one controls the front-panel GPIO connector?
- Every pin on the FP-panel GPIO connector can be connected to a pin on the FP-bank in one of the two radios
- Use the `get_gpio_srcs()`, `get_gpio_src()`, and `set_gpio_src()` APIs to control who is controlling the pins

Concepts & Nomenclature: Sources



- Other USRPs have even more possible sources
- N310 has two channels per radio, each with their own banks
- All embedded devices can control GPIOs from their embedded Linux
- X410 even lets you control the pins from an arbitrary source within the FPGA

Concepts & Nomenclature: Sources/Connectors



- If you have multiple connectors (currently, only on X410) then you can select GPIO sources for every connector. Use the `get_gpio_src_banks()` to enumerate connectors and use the first argument in `set_gpio_src()` to identify which connector you're configuring.

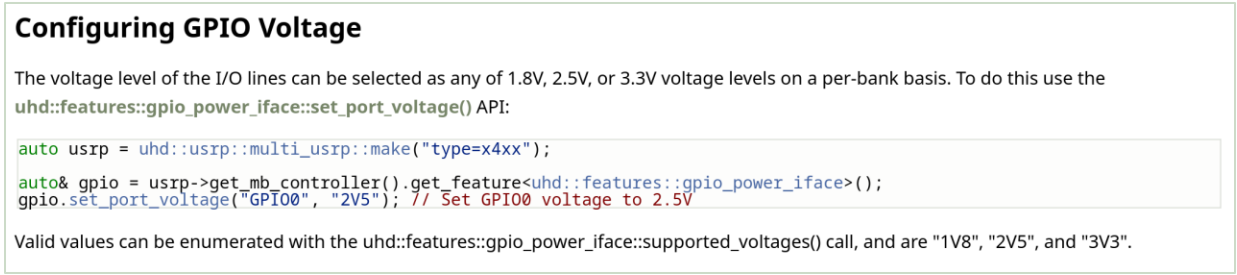
Programming GPIO banks in the FPGA

- When using the regular banks (e.g., FP0 on an X310), pins can be programmed to be:
  - GPIO inputs
  - GPIO outputs
  - ATR pins
- Example: 8 pins controlled by Channel 0
- Bottom 4 GPIOs are ATR, top 4 pins are GPIO
- Pins 6 and 7 are inputs
- ATR pins shall go high when: Pin 0 -> Idle, Pin 1 -> Rx, Pin 2 -> Tx, Pin 3 -> Full duplex
- Pin 5 shall go high 1 second in the future
- We print high/low state of pin 7

```
import time
import uhd
U = uhd.usrp.MultiUSRP("type=x300")
db15_conn_name = U.get_gpio_src_banks()[0]
U.set_gpio_src(db15_conn_name, ['RFA',] * 8 + ['RFB',] * 4)
pin_mask = 0xFF # 8 pins
U.set_gpio_attr('FP0A', 'CTRL',    0xF0, pin_mask)
U.set_gpio_attr('FP0A', 'DDR',     0x3F, pin_mask)
U.set_gpio_attr('FP0A', 'ATR_0X', 1<<0, pin_mask)
U.set_gpio_attr('FP0A', 'ATR_RX', 1<<1, pin_mask)
U.set_gpio_attr('FP0A', 'ATR_TX', 1<<2, pin_mask)
U.set_gpio_attr('FP0A', 'ATR_XX', 1<<3, pin_mask)
U.set_command_time(U.get_time_now() + 1.0)
U.set_gpio_attr('FP0A', 'OUT',     1<<5, pin_mask)
U.clear_command_time()
print(bool(U.get_gpio_attr('FP0A', 'READBACK') & (1<<7)))
```

Topics for another day

- GPIO Voltage API: X410 has a programmable GPIO voltage (1.8, 2.5, 3.3 V)



Configuring GPIO Voltage

The voltage level of the I/O lines can be selected as any of 1.8V, 2.5V, or 3.3V voltage levels on a per-bank basis. To do this use the uhd::features::gpio_power_iface::set_port_voltage() API:

```
auto usrp = uhd::usrp::multi_usrp::make("type=x4xx");

auto& gpio = usrp->get_mb_controller().get_feature<uhd::features::gpio_power_iface>();
gpio.set_port_voltage("GPIO0", "2V5"); // Set GPIO0 voltage to 2.5V
```

Valid values can be enumerated with the uhd::features::gpio_power_iface::supported_voltages() call, and are "1V8", "2V5", and "3V3".

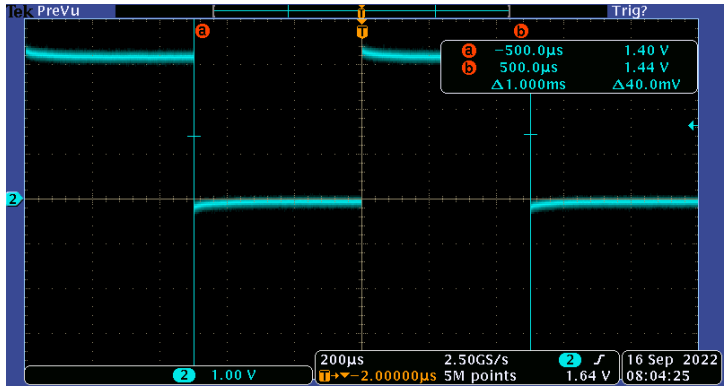
- gpiod API: On embedded devices (E3xx, N3xx, X4xx) GPIOs can be mapped to non-UHD-software-control (using Linux kernel APIs).

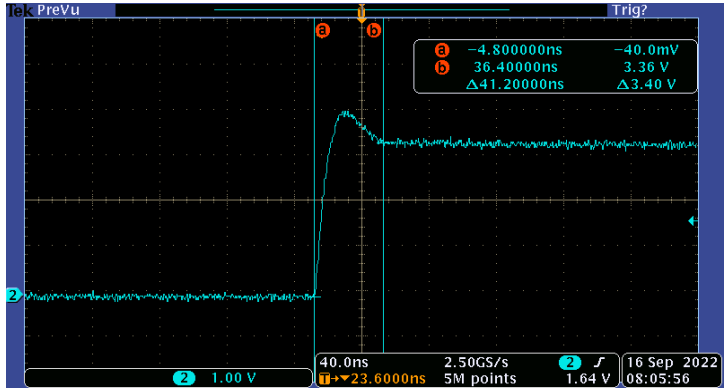
Part 5: Protocols & Signals

Using ATRs

Timed Toggles

- Using the time command API, we can toggle pins at specific times
- This allows us to generate precise clock signals, and toggle all pins in a predictable manner
- => Enables bit-banging, e.g., SPI transactions
- Timing is good, but speed is not (kHz range)
- Note that speed is limited by the circuit design, too (albeit at higher rates)

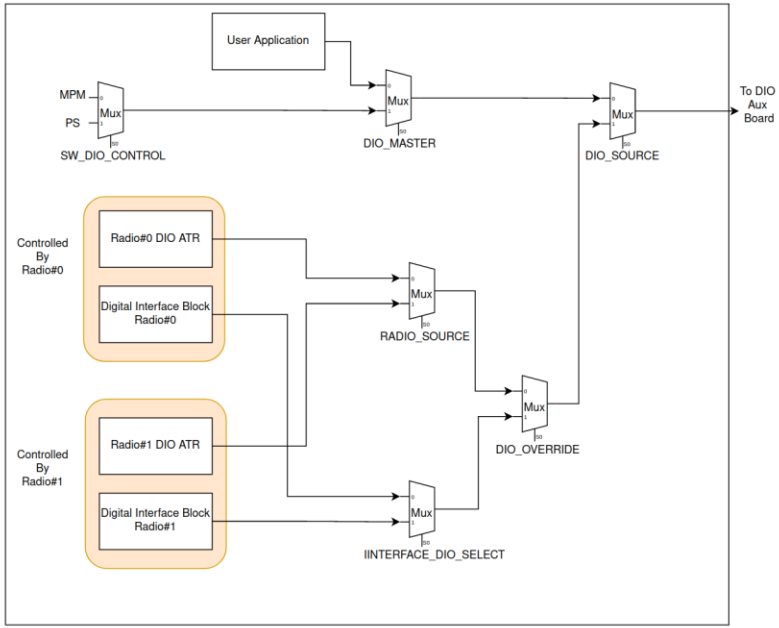





```
import uhd
import time
U = uhd.usrp.MultiUSRP("type=x300")
U.set_gpio_attr('FP0A', 'CTRL', 0x000, 0xFFF)
U.set_gpio_attr('FP0A', 'DDR',  0xFFF, 0xFFF)
pin_state = False
start_time = U.get_time_now() + .5
per_time = 1./1e3 # 1 kHz signal
for i in range(101):
    U.set_command_time(start_time + i * per_time)
    U.set_gpio_attr('FP0A', 'OUT', 0x100, 0xFFF) # Rising edge
    U.set_command_time(start_time + (i + .5) * per_time)
    U.set_gpio_attr('FP0A', 'OUT', 0x000, 0xFFF) # Falling edge
U.clear_command_time()
```

Want fast SPI?

- On X410 only: SPI engine is included on the device!



X4x0 GPIO Source Control

X410 SPI Mode

- See to manual for details on usage
- Allows SPI transactions at up to 100 MHz
- SCLK is integer fraction of radio clock (e.g., run radio clock at 250 MHz, and SCLK at 250/4 MHz = 62.5 MHz)

Configuration of SPI lines

The x4x0 SPI mode supports up to 4 peripherals. All of these peripherals may have a different SPI pin configuration. The pins available for the usage with SPI are listed in **Pin Mapping**. For GPIO0 the available pins are enumerated from 0 through 11, for GPIO1 the available pins are from 12 through 23. The vector of peripheral configurations is passed to the spi_iface_getter to get the reference:

```
uhd::features::spi_periph_config_t periph_cfg;
periph_cfg.periph_clk = 0;
periph_cfg.periph_sdi = 1;
periph_cfg.periph_sdo = 2;
periph_cfg.periph_cs  = 3;
std::vector<uhd::features::spi_periph_config_t> periph_cfgs;
periph_cfgs.push_back(periph_cfg);
auto spi_ref = spi_getter_iface.get_spi_ref(periph_cfgs);

// Set data direction register (set all to outgoing except for SDI)
usrp->set_gpio_attr("GPIOA", "DDR", 0xD, 0xF);
```

Write and read on SPI

With the SPI reference read and write operations can be performed. For doing this, some characteristics of the SPI need to be configured:

```
uhd::spi_config_t config;
config.divider = 4;
config.miso_edge = config.EDGE_RISE;
...
spi_ref->write_spi(0, config, 0xFEFE, 32);
uint32_t read_data = spi_ref->read_spi(0, config, 0xFEFE, 32);
```

The terms 'MISO' and 'MOSI' in the spi_config_t struct map to 'SDI' and 'SDO' respectively. They are legacy terms which will not be used in new code anymore following the resolution to redefine SPI signal names by the Open Source Hardware Association: https://www.oshwa.org/a-resolution-to-redefine-spi-signal-names/

Happy Toggling!

- Thanks for using USRPs!
- Any more questions on GPIOs?