



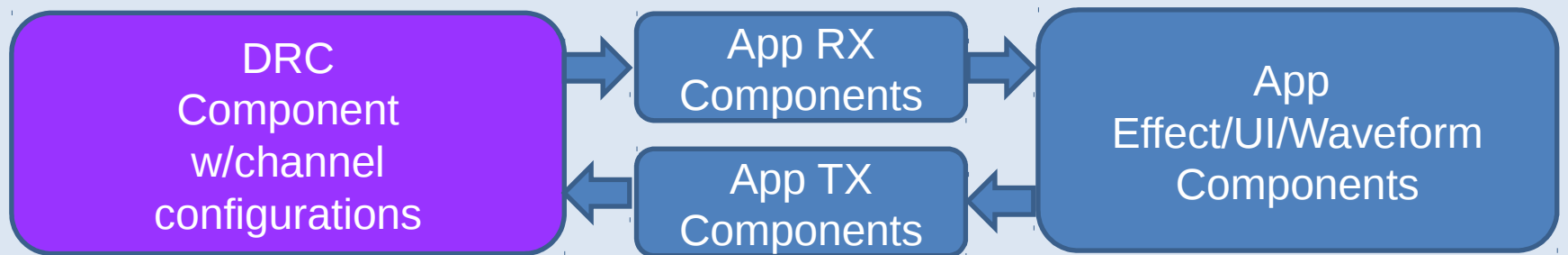
OpenCPI - Open Component Portability Infrastructure

**An Introduction to the
Digital Radio Controller (DRC) model
and APIs, for Controlling and
Configuring RF I/O in OpenCPI Systems
that are “radios” (SDRs)**

Digital Radio Controller (DRC) Software

- OpenCPI runs component-based apps on embedded **systems**
- Some **systems** supported by OpenCPI are “radios”
- Radio systems (Software Defined Radios – SDRs) have **channels** with:
 - RF Antennas at one end and digital samples on the other
- DRC is how OpenCPI applications **control/configure radio channels**
 - E.g. specifying tuning frequency and sampling rates
 - One DRC interface, common across radios.
- Application developers **use** DRC to configure/use RF I/O hardware

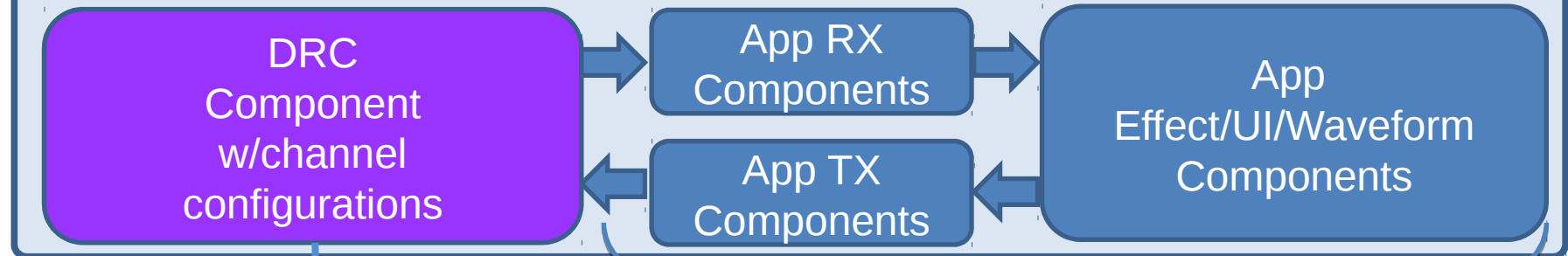
An OpenCPI SDR Application (XML) consists of:



Digital Radio Controller (DRC) Model

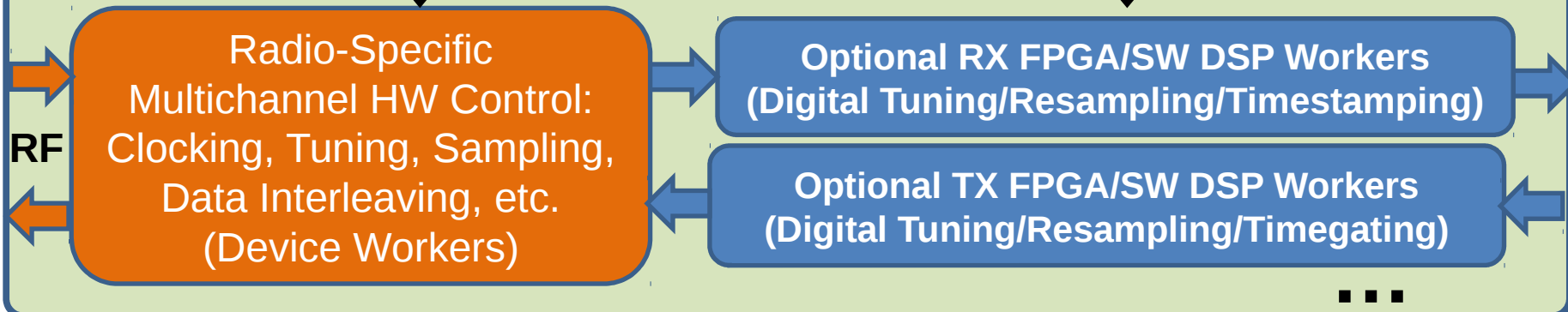
- How OpenCPI SDR Apps run on Radio Systems

OpenCPI SDR Application (XML)



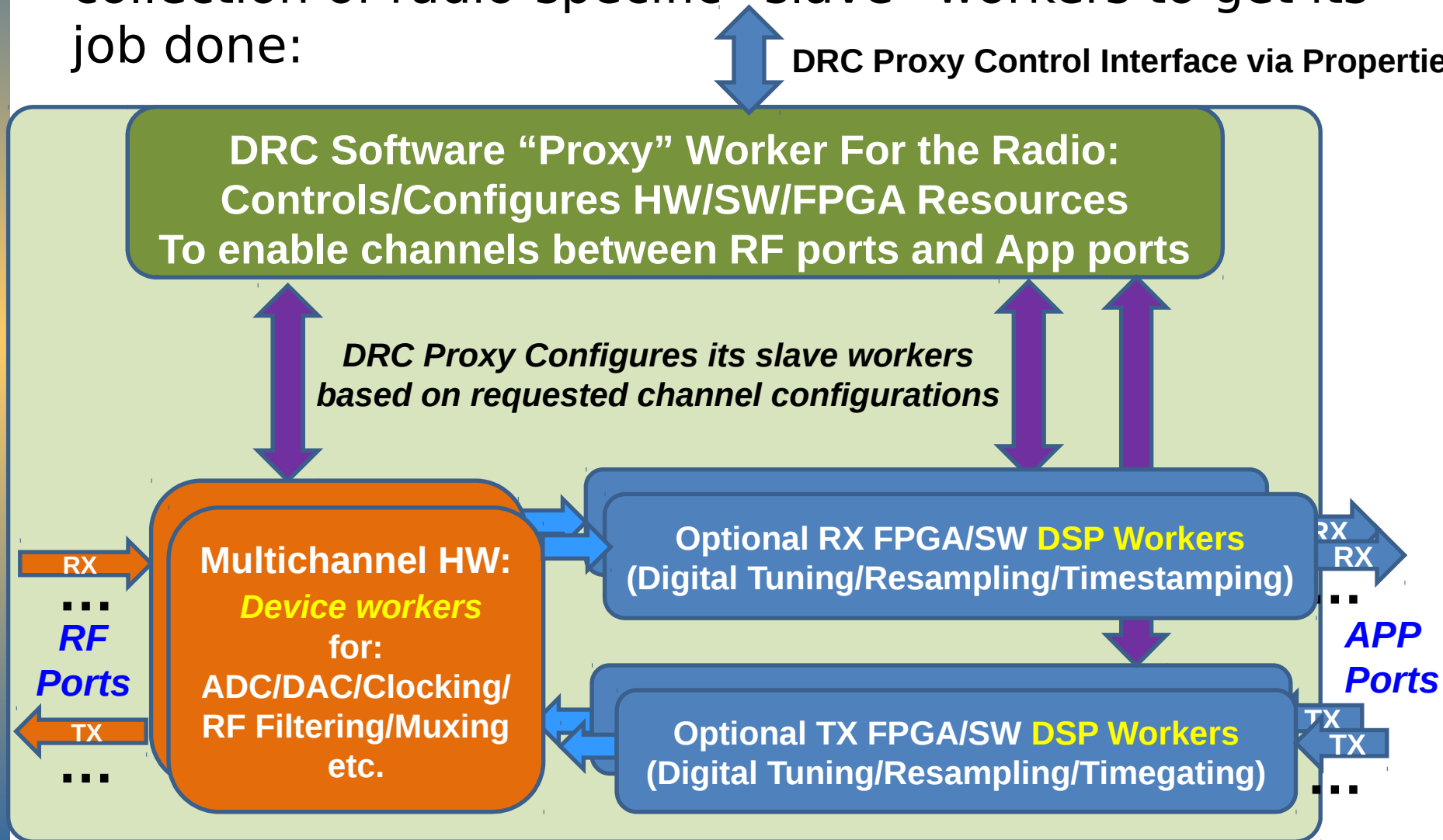
OpenCPI App Components execute using a combination of FPGA and SW workers on CPUs and FPGAs

DRC Software “Proxy” Worker For the Radio:
Controls and Configures Radio/Channel HW/SW/FPGA Resources



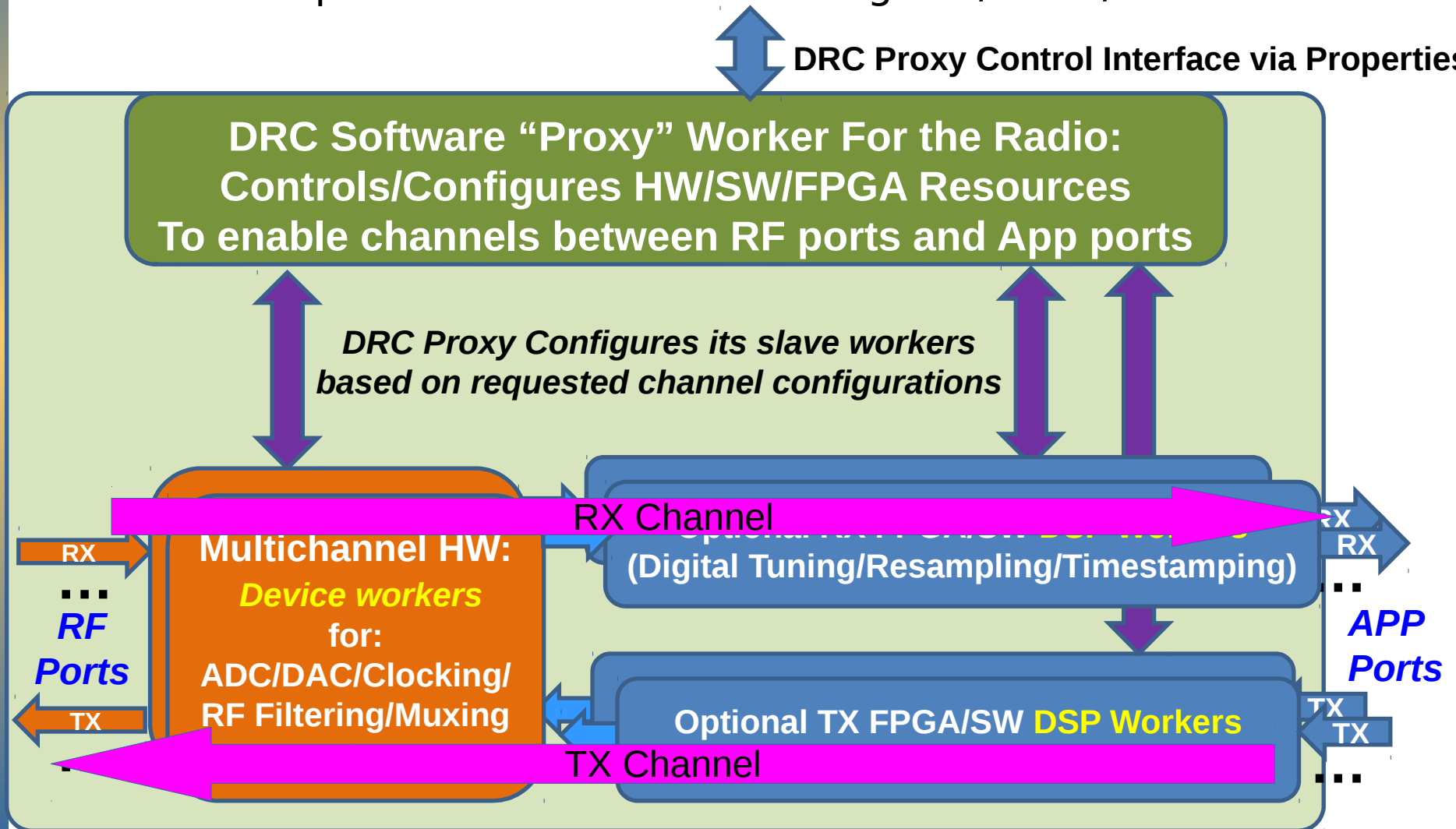
Digital Radio Controller (Proxy) Worker

- The DRC proxy worker (software) *for a radio* uses a collection of radio-specific “slave” workers to get its job done:



Digital Radio Controller *Channels*

- A channel is between an RF port and an App port
- The DRC implements the channel using HW/FPGA/SW resources



- A Previous OpenCPI (pre 1.5) project had a framework group and an SDR applications group
- The apps group identified OpenCPI weaknesses for SDR app portability
- DRC was specified as the solution for portability of SDR apps.
 - *Interface between app developers and platform developers*
- A DRC prototype was developed (~2018/1.5) as a “technology-preview”
 - Only for FMCOMMS3 daughter card with ADI 9361 Xcvr
- Code was modified recently to run on other 9361-based SDRs:
 - E310 and Pluto
- This experience exposed the required work to finish DRC
 - To make DRC supported and suitable for new OSPs, etc.
- Released and supported, as part of 2.1

The DRC Application Interface: XML and



ACI

- A DRC is a **component**, so it has properties and ports
 - DRC **Properties** used to configure channels (between RF ports and App ports)
 - DRC **Ports** (the App ports) convey baseband sample data to/from application components.
- Application XML specifies a DRC instance, e.g.:
 - `<instance component='ocpi.platform.drc'...../>`
- The Application XML connects to the DRC's app ports (rx or tx), e.g.:
 - `<connection>`
 - `<port='rx' instance='drc' />`
 - `<port='in' instance='appcomp1' />`
 - `</connection>`
 - *These app ports are actually directly connected to lower level hardware/FPGA workers*
- One DRC property (**configurations**) specifies one or more **radio** configurations
 - A radio configuration specifies a set of **channel** configurations (to be used together)
 - A radio configuration, when specified in app XML:
 - Is sufficient for static channel configurations

- A channel is RX or TX, and is between two endpoints:
 1. A physical RF (antenna) connector, a.k.a. an **RF Port**
 2. A digital sample “base band” port connected to the app (an **App Port**)
- **RF Ports** are identified by ordinals (of Rx or Tx), but can be named.
 - (RF) connector names are system-specific (labels on connector panel)
- Digital sample **app ports** are numbered (rx[n] or tx[n])
 - A given DRC has enough of these app ports for what is possible.
- A **channel** is a path between an RF connector and the application port
 - But transceivers and radios have RF switching and multiplexing

DRC App Interface: static XML Example

```
<Application>
  <Instance component='drc'>
    <property name='configurations' define one configuration with two channels
      value='{description My static radio channels,
        channels { define two channels, one rx, one tx
          {rx true,
            tuning_freq_mhz 2450,
            bandwidth_3db_mhz 0.24,
            sampling_rate_Msps 0.25,
            samples_are_complex true,
            tolerance_tuning_freq_mhz 0.01,
            tolerance_sampling_rate_msps 0.01,
            tolerance_gain_db 1},
          {rx false,
            tuning_freq_mhz 2450,
            bandwidth_3db_mhz 0.24,
            sampling_rate_Msps 0.25,
            samples_are_complex true,
            gain_db -25,
            tolerance_tuning_freq_mhz 0.01,
            tolerance_sampling_rate_msps 0.01,
            tolerance_gain_db 1}}}'/>
    <property name='start' value='0' /> start first radio config when app starts
  </Instance>
  ... other component instances in the app and connections to app ports ...
</Application>
```

Connections

```
<Application>
  <Instance component='drc' connect='my_rx_processing'>
    <property name='configurations' define one configuration with two channels
      value='{description My static radio channels,
        channels {
          {rx true,
            ...
            tolerance_gain_db 1},
          {rx false,
            ...
            tolerance_gain_db 1}}}'/>
    <property name='start' value='0' /> start first radio config when app starts
  </Instance>
  <Instance component='my_tx_processing' connect='drc' />
  <Instance component='my_rx_processing' />
  ---OR---
  <connection>
    <port name='rx' instance='drc' />
    <port name='in' instance='my_rx_processing' />
  </connection>
  <connection>
    <port name='tx' instance='drc' />
    <port name='out' instance='my_tx_processing' />
  </connection>
</Application>
```

- Static XML Radio Configuration
 - The application's radio configuration is in the Application XML
 - One XML-based configuration can be enabled at app startup
 - This does not preclude dynamic changes later via ACI.
- Dynamic Configurations (possibly more than one)
 - May still be specified in application XML
 - But are not initially enabled
 - Are allowed to conflict since they are enabled separately
 - May also be entirely dynamically specified in ACI/C++ code
 - Enabled/disabled dynamically ACI/C++ code.
- An app may configure some channels statically (on one

- ACI is based on reading and writing properties.
 - Set **configurations** (when not already specified in app XML)
 - Enable/disable configurations
 - Check status of configurations (enabled, disabled, error)
- ACI can query the DRC for constraints and request configurations that may be determined based on what is possible/available.
- ACI may change channel configurations during execution
 - Change frequency/bandwidth/sampling rate/gain etc.
- ACI may change which channels are doing what

Properties

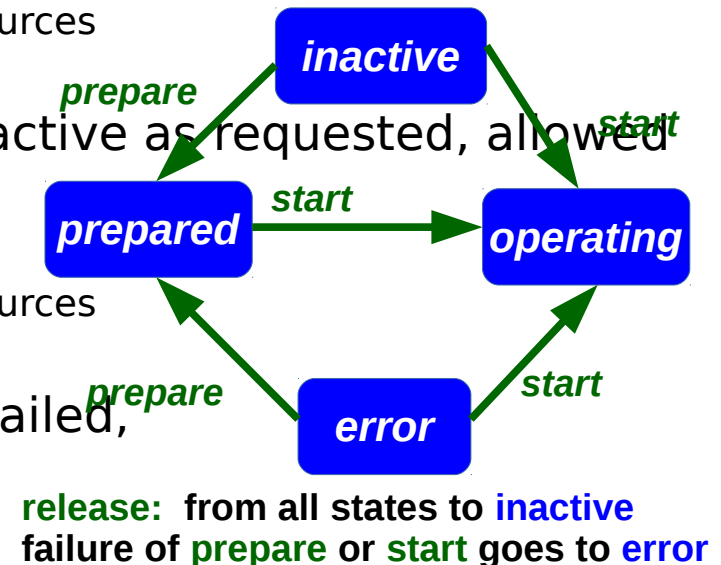
- 1. Configurations:** a sequence of radio configurations
 - Each *radio* configuration is a sequence of *channel* configurations:
 - Which RF Port? Which App Port? RX or TX?
 - Channel attributes: freq/bw/sample rate/gain/real-or-IQ
 - Attribute tolerances
- 2. Status:** dynamic/volatile information about configurations
 - A sequence of configuration status records
 - State: inactive, prepared, operating, error
 - Error: description of why it could not be prepared or started
 - Actual channel attributes/status, within tolerance etc.
- 3. Constraints:** ranges of attribute values
 - A sequence of records, per RF port and direction

- ***Separate properties for each action on a configuration***
- Actions (and thus properties) are:
 - **Prepare** – validate/preprocess configuration
 - Perhaps allocating resources
 - Minimizing “start” latency when started/switched later
 - **Start** – make operational
 - **Stop** – suspend operation of the configuration
 - **Release** — release any resources, make “inactive”.
- Action properties are set with the configuration ordinal as value
- Static starting in app XML is just (for starting configuration 0):
 - `<property name='start' value='0' />`
- Dynamic code example:
 - `OA::Property start(app, “drc”, “start”);`

DRC Radio Configuration Lifecycle:

States

- **Initial configurations** value is a sequence of zero or more configurations
- Configurations can be modified in any state; changes do not take immediate effect.
- Initial state is **inactive**, allowed operations are:
 - **prepare**: verify configuration, allocate resources, get ready to start
 - **start**: (implies **prepare** when state is **inactive**) make the configuration operational
- Next state is **prepared**: ready to be (quickly) **started**, allowed operations are:
 - **start**: make the configuration operational, **operating**
 - **release**: revert to **inactive**, releasing any resources
- Next state is **operating**: channels are active as requested, allowed operations are:
 - **stop**: revert to **prepared** state
 - **release**: revert to **inactive**, releasing any resources
- Error state is **error**: **prepare** or **start** failed, allowed operations are:
 - **prepare, start** or **release**



- If the system is a “radio”, the DRC proxy is part of the OSP
- The DRC proxy manages device workers related to RF I/O
- The DRC translates channel configurations into device-level settings
 - Clock generation
 - RF muxing and switching
 - Hardware-based RF filtering
 - Hardware-based tuning and resampling
- When possible, it manages and detects conflicts between settings
 - I.e. if channel 1 is programmed for X, channel 2 cannot be Y
- When needed, uses DSP workers for tuning/resampling

DRC Layered Architecture (for

developers) *How the DRC is implemented for a radio (notionally).*

- App XML has DRC instance Application Using DRC Component (XML)
- App deployment finds DRC proxy DRC Component Spec (XML)
- DRC proxy written for radio DRC Proxy RCC Worker (C++)
- DRC generic proxy helper code DRC Utility Classes (C++)
- DRC proxy helper for Xcvr/HW DRC Transceiver Class(es) (C++)
- E.g. AD9361 helper library Transceiver Vendor Library (C)
- Control plane to access HW OpenCPI Control Plane
- Device worker written for Xcvr Transceiver Device Worker (XML/VHDL)
- Generic SPI/I2C support OpenCPI SPI/I2C Support
- Actual radio hardware Transceiver Device

Application Dev

OpenCPI Code

Platform Development

Vendor Supplied

DRC Platform Implementation for the

OSP *How the DRC is implemented for a radio.*

- App XML has DRC instance 
- App deployment finds DRC pro 
- **DRC proxy written for radio** 
- DRC generic proxy helper code 
- **DRC proxy helper: Xcvr/HV** 
- E.g. AD9361 NO-OS helper lib 
- Control plane to access HW 
- **Device worker for Xcvr** 
- Generic SPI/I2C support 
- Actual radio hardware 

Digital
Datapath

Application Dev

OpenCPI Code

Platform Development

Vendor Supplied

- Device workers developed and tested without/before DRC support
 - Using direct/explicit configuration properties
 - Should be developed and tested first
- Use OpenCPI Helper Device workers for digital data path
 - Attached to `data_{src,sink}_{q}{adc,dac}`
 - Timestamper, timegate
- Device workers for configuring transceivers (register access)
- Device workers for external clock generators etc.
- Then the DRC proxy XML specifies a slave assembly:
 - Which workers, which properties, how they are connected

- DRC Proxy Worker Implements the DRC Application Interface
 - Based on properties in the generic DRC component spec.
 - DRC is “just another component” in the App.
- There is a helper header file and class for all DRC proxies:
 - **#include “OcpIDrcProxy.hh”**
 - See drc_plutosdr.rcc as an example
- DRC Helper classes & proxies for transceivers/adcdac etc.
 - Some generic helper classes to reduce copy/paste
 - Transceiver code for widely used transceivers, like ad9361
- DRC proxy worker is for the radio system as a whole
 - Multiple transceivers would be integrated in one proxy
 - Separate devices, clock generators, GPIO, etc.
 - Compensating for tuning limitations, per channel

- DRC is a component, with a component specification
- Provides applications access to radio hardware, with ports and properties.
- Implements channels between RF ports and App ports.
- DRC implementations are proxies that use underlying workers to access and control various hardware and DSP elements
- Platform developers create DRC proxies as part of OSPs for radios