Minimalist beamline control and experiment software

LIU Yu <liuyu91@ihep.ac.cn>
Institute of High Energy Physics, Chinese Academy of Sciences, People's Republic of China

NOBUGS 2022 Poster

Introduction

Efforts are being made at the High Energy Photon Source (HEPS) to systematically approach the complexity lower-bounds both in beamline control (DOI: `10.1107/S160057752200337X`) and in experiment software (DOI: `10.1107/S1600577522002697`). This poster gives some representative results, and highlights additional examples which the author believes to have satisfactorily approximated the complexity lower-bounds in certain aspects.

Minimalist beamline control

Our packaging system (GitHub: **`ihep-pkg-ose`**) creates highly reproducible packages for EPICS modules covering the full synApps collection, with support for Rocky Linux 8 added recently. With our Python-assisted inheritance and metadata generation system, adding an EPICS module to our collection is often as simple as is shown below, from which the author believes the complexity lower-bound in EPICS packaging is satisfactorily approximated.

```
# Apart from this .spec file, corresponding entries also need to be added
# to misc/SHA512SUMS/main and misc/pkgs/epics (one-line change for each).
%define repo ADmarCCD
%define commit 8f62ac54
%{meta name license=EPICS github=areaDetector version=2_3,2.commit}
Summary:        EPICS - Rayonix MarCCD detectors
%{inherit ad + deps}
%description
%{inherit ad}
```

EPICS applications involving multiple support modules are traditionally implemented by self-built IOC executables that depend on these modules, but a majority of them can also be done by using multiple single-module IOCs simultaneously. Along with the support modules, provided in our packages are reusable modular IOCs that can be fed with application-specific **`cmd`**s, **`db`**s and ancillary data, which we conventionally put in **`~/iocBoot`** (see below). With extensive use of these IOCs, we can dramatically reduce the need for self-built IOCs, reducing the workload in creating and maintaining the corresponding applications by 1–2 orders of magnitude. This also enables highly maintainable accommodation of most IOCs on a beamline on just a few computers with very modest hardware.

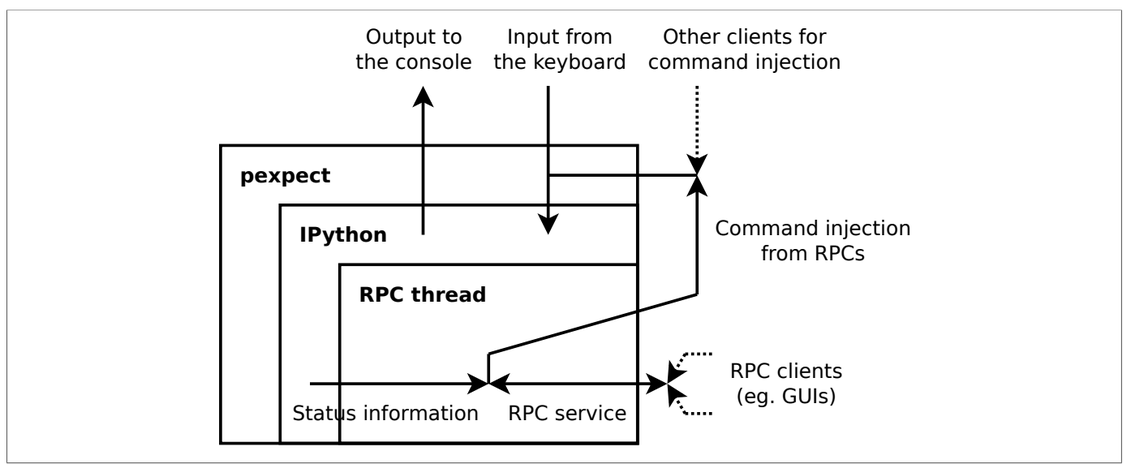
```
#!/bin/sh -e
# Loads config from ${HOME}/iocBoot/${IOC} instead of ${TOP}/iocBoot/${IOC}.
cd /opt/epics/StreamDevice/iocBoot/iocThermo
exec ../../bin/linux-x86_64/streamApp ~/iocBoot/iocThermo/temp18c.cmd

#!/bin/sh -e
# Useful supports, like autosave and iocStats, are added to reusable IOCs.
cd ~/iocBoot/iocSoft
exec softIoc -m 'C=BL3W1:,M=mono:,SCAN=1 second,IOC=BL3W1:Stats:mono:' \
    -d mono.db -d /opt/epics/iocStats/db/iocAdminCore.db

#!/bin/sh -e
# Subdirectories under `iocs' are moved to ${SUPPORT} automatically by
# the packaging system for all motor and areaDetector modules, so no more
# /opt/epics/areaDetector/ADXspress3/iocs/xspress3IOC/iocBoot/iocXspress3.
cd /opt/epics/xspress3IOC/iocBoot/iocXspress3
exec ../../bin/linux-x86_64/xspress3AppQD ~/iocBoot/iocXspress3/st.cmd
```

Efforts are also made to simplify EPICS modules' codebases. For example, the module for Xspress3 contains a lot of code duplicated for each (number of boxes, number of channels) combination configured upstream, most of them generated with the **`iocbuilder`** utility (of thousands lines of code), leading to great difficulty for an outsider to create an IOC for a new combination. The author reduced the amount of these code to a minimum that does not harm clarity (GitHub: **`ADXspress3`**); the code generator, **`xsp3-chan.sh`**, is only about 20 lines of code. A way to configure **`ADXspress3`** is shown below.

```
# Assuming the number of channels is 4.
$ mkdir -p ~/iocBoot/iocXspress3 && cd ~/iocBoot/iocXspress3
$ cp -r /etc/xspress3/calibration/initial/settings cfg-4ch
# Under xspress3IOC/iocBoot are also iocXsp3QD and iocXsp3CARS.
# By using the same technique used in refactoring code from the
# upstream, compatibility with the upstream is ensured at minimised
# cost. (A paper is being written to discuss this technique.)
$ cd /opt/epics/xspress3IOC/iocBoot/iocXspress3
$ cp st.cmd ~/iocBoot/iocXspress3
$ cd ~/iocBoot/iocXspress3 && $OLDPWD/xsp3-chan.sh 4
# Edit ~/iocBoot/iocXspress3/st.cmd: change ${XSP3CARDS}/${XSP3CHANS},
# and change the last $(TOP)/iocBoot/$(IOC) to $(HOME)/iocBoot/$(IOC).
```

Minimalist experiment software

Mamba, our beamline software framework at HEPS, uses Bluesky's **`ophyd`** for device abstraction and **`RunEngine`** for experiment orchestration. To address Bluesky's lack integrated GUIs, Mamba uses command injection and other RPCs, all based on ZeroMQ/JSON, to enable the GUIs and Bluesky-powered command line to complement each other constructively (see below). The HEPS experiment software group is also developing Mamba Data Worker (MDW), a versatile framework to implement full-fledged graphs of modular data pipes, that will meet the diverse needs in beamline experiments.



Experiment parameter generators (EPGs) are a crucial idea in Mamba, which abstract irrelevant or repetitive details on multiple levels: developers, beamline scientists, experiment users. We recently developed an EPG for PandABox-based fly scans (with a paper being written); the author believes that for fly-scan solutions of comparable flexibility, the codebase of this EPG is satisfactorily minimised. It is based on an **`ophyd`** module that implements full control of PandABox's TCP server (through **`pandablocksclient.py`** from **`pymalcolm`**) in less than 400 lines; the code below shows the sequencer table used by the EPG.

```
    table = dict([
        ("trigger", ["POSA>=POSITION", "POSA>=POSITION",
                "POSA<=POSITION", "POSA<=POSITION"]
            if pad > 0.0 else ["POSA<=POSITION", "POSA<=POSITION",
                "POSA>=POSITION", "POSA>=POSITION"]),
        ("position", [lo, hi + pad / 2, hi, lo - pad / 2]),
        ("time1", [live, 0, live, 0]), ("time2", [dead, 1, dead, 1]),
        ("repeats", [num, 1, num, 1]), ("outa1", [1, 0, 1, 0])
    ] + [(f, [0] * 4) for f in seq_outs_not(["outa1"])])
    if not snake:
        table = dict((k, [v[0], v[3]]) for k, v in table.items())
    D.panda = PandaDevice("192.168.1.11", name = "D.panda")
    D.panda.seq1.table.set(table).wait()
    print(D.panda.seq1.table.value.get())
```

Based on the **`ophyd`** module above, our EPG implements constant-speed mapping fly-scans of various dimensions in less than 200 lines, taking care of the configuration of "PandA Blocks" needed for such scans; it is also easily extensible to more advanced fly scans, like those with variable-speed or irregular trajectories. Shown first below is the Bluesky plan provided to users by the EPG, in comparison with its step-scan counterpart; underlying it are modules that can be composed by beamline scientists to implement scans deliberately fragmented to overcome hardware limits: like Xspress3's limit on the number of frames in one exposure series, or PandABox's limit on the number of sequencer entries.

```
# RE.subscribe(mdw_stepscan_callback)
# RE(grid_scan([D.xsp3, D.cnt974a], M.m2, -1, 1, 3, M.m1, -4, 4, 5))
RE.subscribe(mdw_flyscan_callback)
RE(fly_demo(0.5, M.m2, -1, 1, 3, M.m1, -4, 4, 5, period = 0.5))

# def fly_frag(panda, adp, dets, frag_gen, frag_wrap): ...
# def frag_simple(panda, div, duty, *args, period = ..., ...): ...
# def fwrap_adtrig(ads): ...
fly_demo = lambda duty, *args, **kwargs: \
    fly_frag(D.panda, D.adp, [D.xsp3],
        frag_simple(D.panda, 12216 // args[-1], duty, *args, **kwargs),
        fwrap_adtrig([D.xsp3]))

# Currently, additional settings like callbacks, metadata parameters,
# progress notification parameters are injected (or written) like
RE(fly_demo(0.5, M.m2, -1, 1, 3, M.m1, -4, 4, 5, period = 0.5,
    progress = U.progress), cb_gen("fly_demo"), md = U.mdg.read_advance())
# We are now developing further abstractions to encapsulate these code,
# so that the frontend (or the user) may simply run
P.fly_demo(0.5, M.m2, -1, 1, 3, M.m1, -4, 4, 5, period = 0.5)
```