AN EPICS IOC BUILDER

M.G. Abbott, T. Cobb, Diamond Light Source, Oxfordshire, UK

Abstract

An EPICS IO Controller (IOC) is typically assembled from a number of standard components each with potentially quite complex hardware or software initialisation procedures intermixed with a good deal of repetitive boilerplate code. Assembling and maintaining a complex IOC can be a quite difficult and error prone process, particularly if the components are unfamiliar. The EPICS IOC builder is a Python library designed to automate the assembly of a complete IOC from a concise component level description. The dependencies and interactions between components as well as their detailed initialisation procedures are automatically managed by the IOC builder through component description files maintained with the individual components. At Diamond Light Source we have a large library of components that can be assembled into EPICS IOCs. The IOC Builder is further finding increasing use in helping non-expert users to assemble an IOC without specialist knowledge.

INTRODUCTION

The IOC builder is a Python framework designed to automate the process of assembling complex IOCs from existing components. It also provides facilities for generating databases, either as auxiliary records to link components in an IOC, or for the generation of standalone templates.

At Diamond the IOC builder has been used to generate Diagnostics and Power Supply controller IOCs, which show a combination of repetition and variation which is perfect for scripted generation, most area detector IOCs which need complex setting up, and a number of other photon beamline IOCs. The builder has also been used to generate the templates for the Libera EPICS driver [1, 2].

An IOC is normally assembled from existing EPICS support modules together with a small amount of IOC and module specific glue. Input to the IOC builder is either in the form of a structured Python script or a list of component instances specified in XML. The XML version of the IOC builder is designed for more routine IOC definitions where all that really needs to be specified is a list of components provided by EPICS modules and their parameters.

Each EPICS support module at Diamond can be found in a well defined location and has a clearly defined structure. As part of this structure IOC builder definitions are defined by each support module; these define the components such as templates or hardware drivers that go into an IOC.

```
import iocbuilder
iocbuilder.ConfigureIOC()
from iocbuilder import *

# Load support module definitions for needed modules
ModuleVersion('ipac',         '2-8dls4-5')
ModuleVersion('Hy8515',       '3-9')
ModuleVersion('asyn',         '4-10')
ModuleVersion('streamDevice', '2-4dls2-1')
ModuleVersion('newstep',      '1-4')

# Define the hardware resources used
card4 = modules.ipac.Hy8002(4)
serial = card4.Hy8515(0)

# Create two newstep controllers
for ch in range(2):
    asyn = modules.asyn.AsynSerial(serial.channel(ch))
    modules.newstep.NSC200(
        M = 'TS-TEST-DEV-01', P = '',
        PORT = asyn.DeviceName(), CH = ch)

WriteNamedIoc('ioc', 'TEST-IOC')
```

Figure 1: Complete script to assemble a simple IOC with serial hardware and stream device protocol definitions.

BUILDING AN EXAMPLE IOC

Figure 1 shows a complete IOC builder script for assembling a simple IOC, which in this case uses the streamDevice support module to communicate over a serial port provided by the Hytec 8515 IP card. Here we need five support modules with specific versions loaded by the calls to `ModuleVersion` and two items of hardware (the 8515 serial IP module and a carrier card in which to install it), and we create two instances of the newstep NSC200 component before writing out the entire IOC to the `ioc` subdirectory.

The IOC that is written out contains a top level make file in standard EPICS style together with a standard `configure` directory containing a file `RELEASE` with the relevant `ModuleVersion` definitions converted into EPICS version definitions. The rest of the IOC directory contains the necessary makefiles to link all the required libraries and EPICS dbd files together and template substitutions to complete the definitions.

Note that inter-module dependencies are automatically handled, the only thing the user needs to do here is to specify module versions in the correct order. The XML builder even automates this step by taking as input an existing `configure/RELEASE` file and walking the associated dependency tree.

USING THE IOC BUILDER

An IOC build script normally has a fairly stereotypical structure, consisting of four stages:

- Initialisation and configuration.
- Loading module definitions.
- Creation of IOC resources.
- Generation of output.

Initialisation and Configuration. The IOC builder can be used to generate IOCs to target a variety of architectures; at Diamond we support VxWorks, Linux and Windows. Alternatively the builder can be used to just generate a record template file. To choose between these options the builder must be configured before importing most of the iocbuilder symbols by calling one of the functions `ConfigureIOC` or `ConfigureTemplate`.

Loading Module Definitions. Each EPICS support module at Diamond contains IOC builder definitions specific to that module in the file `etc/builder.py`, and these definitions must be loaded into the builder by calling the builder `ModuleVersion` function before the support module can be used. Many support modules have dependencies on other support modules, so these dependencies must be loaded in the correct sequence.

Creation of IOC Resources. At its simplest an IOC can be defined as just a list of component instances where each component is a resource defined by a support module. The XML builder uses this approach where the input to the builder is essentially a table of component names and their arguments.

More generally it is sensible to structure an IOC definition script with a little bit of care. Hardware definitions can be placed first and it makes sense to separate them from the software resources that use them. Any required resource initialisation will automatically be created as required.

All the resources defined by individual support modules are loaded as Python submodules of the module `iocbuilder.modules` as can be seen in Figure 1.

Generation of Output. Typically the IOC builder can either write out a complete Diamond conformant IOC directory structure, or a single template file, using the functions `WriteNamedIoc` or `WriteRecords` as appropriate.

Creating EPICS Record Templates

The IOC builder was originally written to create complex databases, as gluing together hand-written templates with substitutions or macro processing rapidly becomes cumbersome. This functionality is available for normal IOC generation when some EPICS records are required to connect components together, but is more useful for building complex EPICS record templates.

The basic building block for doing this is the `iocbuilder.records` object which has record constructing methods for each record type. This object is automatically populated as DBD files are loaded, both from EPICS base on startup and from individual module definitions. Records are constructed in memory as Python objects with attributes for each possible record field, and assignments to these fields are automatically checked against the DBD definitions.

To simplify the naming of records when creating them a configurable record naming convention is established when initialising the IOC builder, both enforcing the Diamond record naming convention, and allowing the prefix of the name to be established separately.

Frequently records are generated in association with hardware with specific values written to the `DTYP` and `INP` or `OUT` fields. The builder supports this with record constructors tied to hardware instances which automatically populate these fields.

Figure 2 shows a fragment of a script using these mechanisms to generate a database with 22 record and 217 field definitions.

```
SetChannelName('SA')
power = Libera.ai('POWER',
    LOPR=-80, HOPR=10, ESLO=1e-6, EGU='dBm', PREC=3,
    DESC = 'Absolute input power')
current = Libera.ai('CURRENT',
    LOPR=0, HOPR=500, ESLO=1e-5, EGU='mA', PREC=3,
    DESC = 'SA input current')
Trigger(False, ABCD_() + ABCD_N() + XYQS_(4) +
    [power, current] + MaxAdc())
UnsetChannelName()
```

Figure 2: Fragment of script to generate an EPICS template. Here `Libera` is a device instance, `Trigger` creates a fanout record, and its arguments are functions for generating further records.

Using the IOC builder with an XML file

While the Python interface is useful for building many similar IOCs, many of our IOCs take the form of hundreds of similar objects with slightly different parameters, e.g. motors with different resolutions. This can be better presented in a tabular form, so the IOC builder contains a utility called the XML builder that can build an IOC from parameters stored in an XML file and an interactive graphical editor `xeb` ("XML Editor for the Builder", see Figure 3) that can read and write the correct format XML file. To support this, each argument of every builder object must be annotated with a description and a type as described in the `ArgInfo` section below. The editor can then do suitable error checking and type casting of the arguments, and present descriptive prompts to the user.

In fact, the XML builder and Python scripting are two equivalent approaches with similar power that suit different working styles.

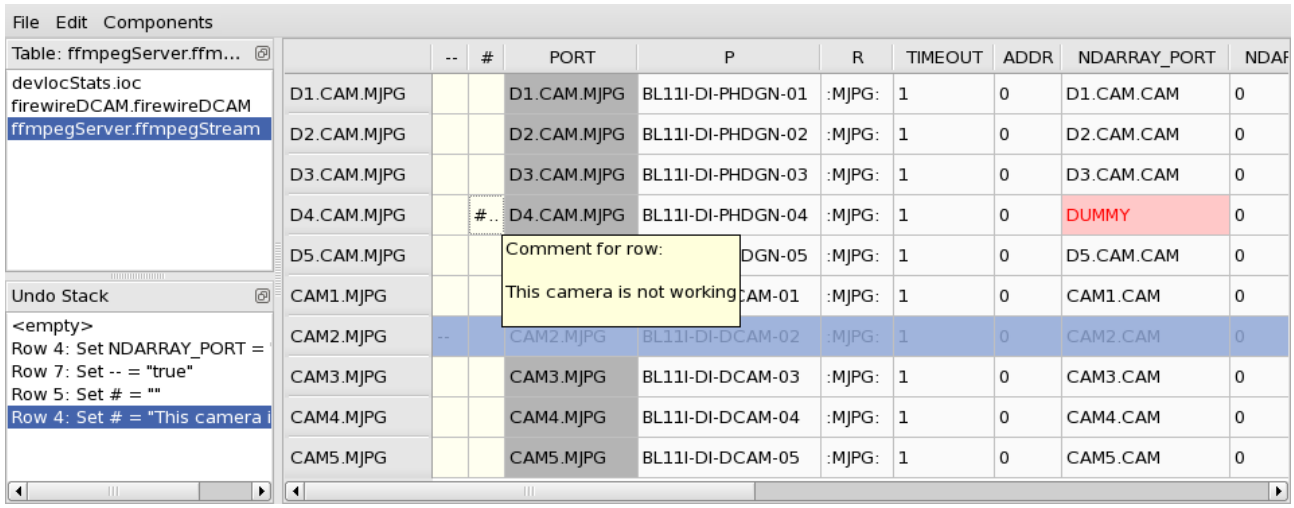

Figure 3: XEB: XML Builder Editor showing camera configuration for an area detector IOC.

MODULE DEFINITIONS

For an EPICS support module to be useful with the IOC builder it is necessary to create IOC builder class definitions for the resources defined by the module. These should encapsulate any module, linkage and DBD dependencies, and should also define any startup script commands needed by the support module. Ideally a good set of builder definitions should capture all the specific knowledge required to use the support module so that the developer of the IOC doesn't need to know.

The builder definitions for a support module are loaded from the file `etc/builder.py` in the base of the support module, which is loaded as a submodule of `iocbuilder.modules`. Builder resources are generally defined by creating subclasses of the `ModuleBase` class exported by the builder, largely as subclasses of `Substitution` to declare EPICS database templates, or of `Device` to declare resources with linkage or initialisation dependencies.

A support module builder definition file should open with imports from `iocbuilder.modules` of all support modules on which it depends to ensure that all dependencies have been loaded.

Module Base

Component resources are normally declared as subclasses of `ModuleBase`. This class performs a number of important functions:

- Ensures the module is included in the build.
- Ensures any module dependencies are resolved, this includes ensuring that dependent modules are linked and initialised in the correct order, and may involve automatically loading other modules.
- Defines path dependent access to files defined by the module. For example, this is used to locate the template file in a `Substitution` definition.

Any subclass of `ModuleBase` can define the value `Dependencies` to specify a list of other modules which must be instantiated before this class is used.

```
class NSC200a(Substitution):
    Arguments = ['P', 'M', 'CH', 'PORT']
    TemplateFile = 'NSC200.template'

    ArgInfo = makeArgInfo(
        P    = Simple('Device Prefix', str),
        M    = Simple('Device Suffix', str),
        CH   = Simple('Channel number', int),
        PORT = Simple('Asyn port string', str))

class NSC200b(AutoSubstitution):
    TemplateFile = 'NSC200.template'
```

Figure 4: Template definition examples showing both manual and automatic template definitions.

Template Definitions

Template definitions are the simplest to write, and are defined as subclasses of the `Substitution` builder class. The defined subclass must define the template file name, an enumeration of the arguments required by the template, and their descriptions. The definition of `NSC200a` in Figure 4 shows this.

The constructor for `NSC200a` will then expect to be called with exactly the listed keyword arguments and will fail otherwise. An alternative constructor can be defined for the class if appropriate.

The class `AutoSubstitution` is a subclass of `Substitution` which automatically searches the specified template file for template parameters to populate the `Arguments` and `ArgInfo` fields, and so allows a template definition to be just two lines as show in the definition of `NSC200b` in Figure 4; this is equivalent to `NSC200a`.

Every instance of a `Substitution` subclass results in the appropriate entries in the `Db/Makefile` and an associated `.substitutions` file.

Device Definitions

The `Device` class is used for component resources which need any of the following elements:

- Startup script initialisation.
- Loading of DBD files, either for new record definitions or for record device type definitions.
- Loading of libraries.
- Custom entries in Makefiles.

A component resource is created by defining a subclass of `Device` and setting values for a number of values, including:

LibFileList A list of libraries that must be linked when using this component.

DbdFileList A list of DBD files defining the EPICS resources provided by and making up this component.

Initialise() This method will be called during the generation of the IOC startup script to generate any device specific initialisation code.

Typically every support module should have a `Device` definition which defines `LibFileList` and `DbdFileList` to record the libraries and DBD files used to load the support module and which is marked as a dependency of the other components of the support module.

Component Argument Descriptions

The XML builder expects every builder component to be annotated with "metadata" documenting the parameters required to instantiate that component. This is done by creating an `ArgInfo` object, for example as shown in the definition of `NSC200a` in Figure 4.

`makeArgInfo` optionally takes the `__init__` method as the first argument, if a custom `__init__` method has been defined, followed by named arguments describing each argument that should be passed to `__init__`. Each of these arguments can be one of:

Simple A simple type

Ident An identifier, lets you specify that this argument should be something of a particular type

Choice One of a list

Enum As choice, but pass the index of the selection to the `__init__` method

It is also possible to add ArgInfo objects together, and filter them using the `filtered` method. This allows more complicated argument structures to be built up.

The `AutoSubstitution` simplifies this process for templates: as the template file is scanned for arguments, an `ArgInfo` object is automatically assembled at the same time. Specially formatted comments in the template file can be used to provide argument descriptions.

```
class Hy8002(IpCarrier):
    MaxIpSlots = 4      # 4 IP slots in this card

    def __init__(self, slot, intLevel=2):
        self.__super.__init__(slot)
        self.intLevel = intLevel

    ArgInfo = makeArgInfo(__init__,
        slot     = Simple('VME Slot number', int),
        intLevel = Simple('VME Interrupt Level', int))

    @classmethod
    def UseModule(cls):     # Assign shared interrupt
        super(Hy8002, cls).UseModule()
        cls.swapint = cls.AllocateIntVector()

    def Initialise(self):
        self.InitialiseCarrier(
            'EXTHy8002', self.slot, self.intLevel,
            self.swapint)
```

Figure 5: Module definition for Hy8002 IP carrier card.

An Example Device Definition

Figure 5 shows the complete module definition for the Hy8002 IP carrier device, part of the ipac support module. The `IpCarrier` class is a subclass of `Device` providing special support for IP carrier cards.

This device is fairly unusual in having only two arguments, but typical in the use of defaults. The `UseModule` method is called by `ModuleBase` immediately before creating the first instance of this class.

CONCLUSIONS

The IOC builder is being used for an increasing number of IOCs at Diamond and has been under continuous development for around five years by both authors.

There are some obstacles to using the IOC builder outside Diamond, the largest being the dependency on the Diamond directory structure. The builder code is reasonably well structured, though a transition to documentation using Doxygen seems to have been unhelpful.

Work on the builder continues in response to internal Diamond developments.

This work was first presented outside Diamond at the EPICS workshop at ICALEPCS 2009 [3].

REFERENCES

[1] M.G. Abbott, G. Rehm, I.S. Uzun, "The Diamond Light Source Control System Interface to the Libera Electron Beam Position Monitors", ICALEPCS 2009.

[2] `http://controls.diamond.ac.uk/downloads/other/libera`

[3] M.G. Abbott, "EPICS IOC builder", EPICS collaboration meeting, ICALEPCS 2009, `http://kds.kek.jp/contributionDisplay.py?contribId=16&confId=3834`.