WHATRECORD: A PYTHON-BASED EPICS FILE FORMAT TOOL *

Kenneth Lauer†, SLAC National Accelerator Laboratory, Menlo Park, CA

Abstract

`whatrecord` is a Python-based parsing tool for interacting with a variety of EPICS (Experimental Physics and Industrial Control System) file formats, including V3 and V7 database files. The project aims for compliance with epics-base by using Lark grammars that closely reflect the original Lex/Yacc grammars.

`whatrecord` offers a suite of tools for working with its supported file formats, with convenient Python-facing dataclass object representations and easy JSON (JavaScript Object Notation) serialization. A prototype backend web server for hosting IOC (Input/Output Controller) and record information is also included as well as a Vue.js-based frontend, an EPICS build system `Makefile` dependency inspector, a static analyzer-of-sorts for startup scripts, and a host of other things that the author added at whim to this side project.

BACKGROUND

The Problem - and the Inspiration

Before digging into the details of the `whatrecord` [1], toolsuite, let us first take a look at the problem and the inspiration behind its creation.

At the LCLS (SLAC's Linac Coherent Light Source), the accelerator and photon side control systems include approximately 3000 IOC instances in total, with hundreds of modules and dozens of versions per module.

In general, these EPICS [2] IOCs, modules, and extensions are comprised of a conglomeration of unique file formats. Some common examples of such file formats include:

- Process database files (`.db`)
- Database definition files (`.dbd`)
- Template / substitutions files
- IOC shell scripts (`st.cmd`)
- StreamDevice protocols (`.proto`)
- State notation language programs (`.st`)
- Gateway configuration (`.pvlist`)
- Access security files (`.acf`)
- Build system `Makefiles`

Additionally, facility-specific tools (centralized IOC management tools like LCLS's IOC Manager, archiver appliance automation tools, and so on) build on top of IOCs and records.

Combined, this makes for an enormous code base with a mix of these EPICS-specific file formats.

Links between these files are often implicit. Take, for example, that an EPICS IOC record has a specific record type name alongside its name in a database file (`.db`), an EPICS PV (Process Variable) name, in a traditional IOC, starts with the record name defined in a database file. This PV name acts as a global identifier that allows for clients on the same network subnet to access - and potentially modify - related data.

A record is made up of fields which can contain metadata like engineering units or user-specified descriptions, references to other records, relevant data values, and so on.

An example record instance, defining a single AI (analog input) record named `IOC:RECORD:NAME` is as follows:

```
record(ai, "IOC:RECORD:NAME") {}
```

This file does not define what the fields of the record type; that is the responsibility of the database definition file (.dbd). A simplified excerpt from a database definition file, defining a single field for the "ai" record type is as follows:

```
recordtype(ai) {
    ...
    field(NAME, DBF_STRING) {
        special(SPC_NOMOD)
        size(61)
        prompt("Record Name")
    }
    ...
}
```

Note that there is no explicit link between the database file and the database definition file: neither reference the other by filename. Rather, one can only infer the link by examining a third file, the IOC-specific IOC shell script (.cmd) file, line-by-line.

An excerpt from such a startup script could look like:

```
dbLoadDatabase("path/to/the.dbd",0,0)
IOC_registerRecordDeviceDriver(pdbbase)
dbLoadRecords("records.db")
```

Each line of this script includes up to one command. Each of those commands has been registered by either EPICS itself, the modules included in the IOC, or the IOC source code itself. Typically, the available commands would be found either in documentation or by executing the IOC and invoking the built-in help system. Alternatively, the most reliable fallback ends up being the source code itself.

Other direct or indirect references may be found inside fields. For example, depending on the `DTYP` (device type) field, the `INP` (input specification) field may be a custom string defined at the device support layer. Interpretation of this field requires knowledge of how these are formatted. Take StreamDevice [3], a generic support module for communicating with controllers that use simple byte streams for communication, for example:

```
record(ai, "IOC:RECORD:NAME") {
```

* Work supported by U.S. D.O.E. Contract DE-AC02-76SF00515.
† klauer@slac.stanford.edu

```
    field(DTYP, "stream")
    field(INP, "@ProtocolFilename.proto getValue
        PS1")
}
```

The device type here is set to `"stream"`, a custom identifier that StreamDevice has hard-coded. This instructs EPICS to use StreamDevice and interpret the `INP` field with it. It us up to the IOC developer to understand the format of these strings and set them appropriately, in order to reference back to the protocol file that defines the byte string to send and the expected response format. Here, a StreamDevice protocol file for the above record indicates that a simple string `WHAT:IS:THE:VALUE?` is to be sent, and a floating point value (`%f`, as in the C `scanf` format specifiers) is to be sent from the controller:

```
getValue {
    out "WHAT:IS:THE:VALUE?";
    in "%f";
}
```

This section is a small but important part of what makes up an IOC: the build system surrounding all of these files, other modules with their own standards, access security configuration for intra-subnet access, gateway configuration controlling inter-subnet access, facility-specific tools that rely on PV names, and so on further complicate the number of files and references one needs to be aware of.

While those familiar with EPICS IOC development may find that the above is obvious and simple, it can be opaque at best to those unable to dedicate the time to reading through esoteric (and often outdated) manuals or source code.

Goals and the Emergence of whatrecord

The previous section's problem led the author over the years to desire a tool that could somehow unify these file formats and provide the ability to inspect the links.

These initial goals led to the creation of this new Python package, `whatrecord`:

- Allow for easy parsing of all the special formats outside, and represent them in a widely-used interchange format like JSON.
- Aid the user in the understanding of existing IOCs, whether they are deployed and running or not.
- Provide a method to see how different records, different IOCs, all relate to one another, without requiring the IOC to be running.
- Provide a method for cross-referencing a PV name to its database file, record definition, startup script, and IOC.

With these implemented, pathways for new possibilities were opened: the ability to linking records to PLC code, to StreamDevice protocol information, to gateway access rules, and even shell commands to their respective source code.

CORE FUNCTIONALITY

Overview

`whatrecord` will parse any of the following into intuitive Python dataclasses using the Lark [4] parsing toolkit:

- Database files (V3 or V4/V7), database definitions, template/substitution files
- Access security configuration files
- Autosave .sav files
- Gateway pvlist configuration files
- StreamDevice protocol files
- snlseq/sequencer state machine parsing

IOC shell scripts (i.e., `st.cmd`) can be interpreted and annotated with contextual information during the loading process. The process aims to record what files were loaded during startup, what records will be available in the IOC, what errors were found when loading, what file and line did each record get loaded, and what are the inter- or intra-IOC record relationships.

Additionally, `whatrecord` offers tools for:

- Exporting all parsed results to JSON-serializable objects.
- EPICS build system `Makefile` introspection, a sumo [5]-inspired implementation.
- GDB Python script that inspects binary symbols to find IOC shell commands, variables and source code context

```
dbLoadRecords [str: fname] [str: subs]
.../src/ioc/db/dbIocRegister.c:53
```

- Accurate EPICS macro handling using epicsmacrolib [6].
- Linting startup scripts.
- Plugins for loading happi devices, TwinCAT PLC projects, and IOC information from LCLS's IOC manager.
- Process database record to Beckhoff TwinCAT PLC source code definition mapping (when used in conjunction with pytmc [7]).

An intuitive Python API, user-facing command-line tools, a web-based API/backend server to monitor IOC scripts and serve IOC/record information, and a Vue.js-based frontend single-page application are also provided.

Parsing with Lark

whatrecord utilizes the Lark parser internally to parse most of its supported file formats. Lark supports writing custom parsers with EBNF (extended Backus–Naur form) metasyntax. It is capable of parsing all context-free grammars with ambiguity resolution.

The grammars implemented in whatrecord closely resemble those in EPICS because the source Lex/Yacc grammars are syntactically similar. A test suite is included in whatrecord which attempts to cover various aspects of the packaged grammars.

Parsing a file in whatrecord results in user-friendly type annotated dataclass instances. These can be readily inspected programmatically, serialized to JSON, and - in several cases - exported back into their original format.

```
import whatrecord
db = whatrecord.parse(
    "whatrecord/tests/iocs/db/basic_asyn_motor.db"
)
record = db.records["$(P)$(M)"]
print(record.fields["TWV"].value) # -> '1'
```

A key feature of whatrecord's parsing utilities is that it records the context of many operations. When two different files are used to load a record, both will be present in the context information:

```
import whatrecord
ioc = whatrecord.parse(
  "whatrecord/tests/iocs/ioc_a/st.cmd"
)
db = ioc.shell_state.database
record = db["IOC:KFE:A:One"]
print(record.context)
```

Yields the following, indicating the files and line numbers used to load the given database record instance:

```
(whatrecord/tests/iocs/ioc_a/st.cmd:13,
 whatrecord/tests/iocs/ioc_a/ioc_a.db:1)
```

Exporting/Serializing to JSON

With the whatrecord command-line entry point, you can parse supported formats and pipe their information to tools like jq [8] to interact and run basic queries on data.

The following lists all records in a database file and selects a set of information:

```
$ whatrecord parse
    whatrecord/tests/iocs/db/pva/iq.db |
   jq '.records[] | [.name, .record_type,
       .fields.OUT.value]'
[
  "$(PREFIX)Rate",
  "ao",
  "$(PREFIX)dly_.ODLY NPP"
]
[
  "$(PREFIX)Delta",
  "ao",
  null
]
...
```

The following inspects some EPICS V4 Q:Group settings:

```
$ whatrecord parse
    whatrecord/tests/iocs/db/pva/iq.db |
   jq '.records[] | [ .name, .info["Q:group"]]'
[
  "$(PREFIX)Rate",
  null
]
[
  "$(PREFIX)Phase:I",
  {
    "$(PREFIX)iq": {
      "phas.i": {
        "+type": "plain",
        "+channel": "VAL"
      }
    }
  }
]
...
```

FORMATS AND IMPLEMENTATION NOTES

Database Files whatrecord implements two grammars for EPICS process databases, as there are significant differences between the grammars used for EPICS V3 and V4+ IOCs.

A flag is available --v3 for whatrecord parse to aid facilities that have not yet opted in to using V4+.

Access Security Configuration Files (ACF) whatrecord parses ACF files that target typical EPICS IOCs along with the EPICS gateway. This information can be correlated to records in the frontend, covered in a later section.

Substitution Files Substitutions files and those supported by dbLoadTemplate are readily supported by whatrecord. Contextual information as to which files are loaded in the process is reliably carried along.

A separate grammar for the format used by the command-line tool MSI (the Macro Substitution and Include tool) is also provided due to their differing implementations.

Autosave Save Files Autosave save files (.sav), which track the state of a record such that it may be restored at the next boot of an IOC, are supported by whatrecord. In the web frontend, these values will be displayed alongside those in the database file.

Gateway PV Lists Gateway `.pvlist` files are supported by `whatrecord`. In the frontend, this allows for the easy correlation of gateway rules to records that apply to it, and also in reverse, showing which gateway rules apply to the selected record.

Sequencer - State Notation Language The EPICS sequencer's state notation language format `.st` files is supported by `whatrecord` with some caveats. `whatrecord` does not have full support for a C preprocessor, which the sequencer relies on. This means that for simple files (and those that only use simple C `defines`), `whatrecord` successfully parses the files, whereas complicated `defines` could fail to parse with the included grammar.

LCLS-Specific Formats At the LCLS, there are additional file formats that are in common use. The `epicsArch` format enumerates process variables that are to be included for recording in its data acquisition system. `whatrecord` provides support for this format and allows for linting and display through the frontend.

The LCLS photon controls team uses the Python suite Bluesky [9] for slow-speed data acquisition, with approximately 1,000 Ophyd devices indexed in a happi [10] database. `whatrecord` provides happi integration, mapping process variables used in Ophyd devices back to their IOCs.

Startup Scripts Startup scripts do not have a corresponding grammar. In the EPICS implementation, the IOC shell parses commands character-by-character with a custom routine. `whatrecord` has a ported version of this parsing function for maximum compatibility, supporting redirection and everything else from the original.

Macro Handling Files supported by `whatrecord` often use the EPICS macro library (macLib) in order to interpolate variables (environment variables or otherwise) to their corresponding values.

`whatrecord` uses epicsmacrolib, which is a Cython-wrapped version of EPICS macLib, under the hood to dutifully reproduce standard EPICS macro expansion and state tracking.

Makefiles `Makefiles` can be inspected to reveal per-module or per-IOC dependencies and settings of the EPICS build system. Unlike the other supported formats, `Makefiles` are not parsed but rather introspected by way of GNU `make`, an optional external requirement.

The implementation is noted as sumo-inspired, as an important part of how sumo scans source code is replicated. The library executes `make` with a custom target, enabling the direct handling of any and all `Makefile` syntax that the host supports.

`whatrecord` is able to extract a variety of information from a `Makefile` in this manner, exporting information such as the EPICS build architectures, cross-compiler host architectures, target architectures, the EPICS base version, configuration paths, release top variables, environment variable settings, and so on.

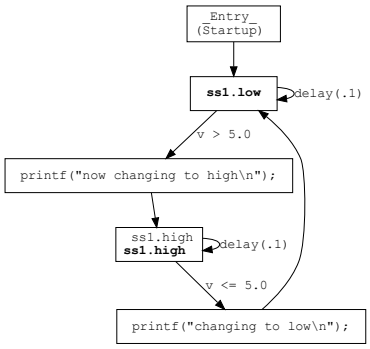

Figure 1: A sample state notation language program graph.

Backend Server

Building on top of the parsing tools, `whatrecord` provides an aiohttp [11]-based "backend" server daemon that enables users to query IOC-related information over a RESTful (REpresentational State Transfer) JSON interface.

A summary as to how the backend server operates is as follows:

1. Find all EPICS IOCs specified by the user (or those listed in LCLS's IOC manager tool).

2. Load the startup scripts with the built-in parsing tools, including databases and supported files.

3. Periodically check previously-loaded files for changes, and re-load IOCs.

4. Listen for clients (or the `whatrecord` frontend) querying for IOC information by way of aiohttp.

Command-Line Tools

whatrecord Deps This tool utilizes the `Makefile` parsing tools internally to recursively generate a dependency graph of supporting modules to a given IOC.

whatrecord Graph This graphing tool allows for graphing of records or state notation language transition diagrams. Inter-IOC record links may be graphed if multiple database files or startup scripts are specified.

A sample state notation language graph is shown in Fig. 1, and a sample record relationship graph is shown in Fig. 2.

whatrecord Server This command spawns the backend server. It may also be used to export a cached state for offline usage by the frontend.

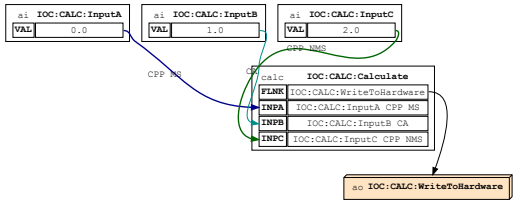

Figure 2: A sample record relationship graph.

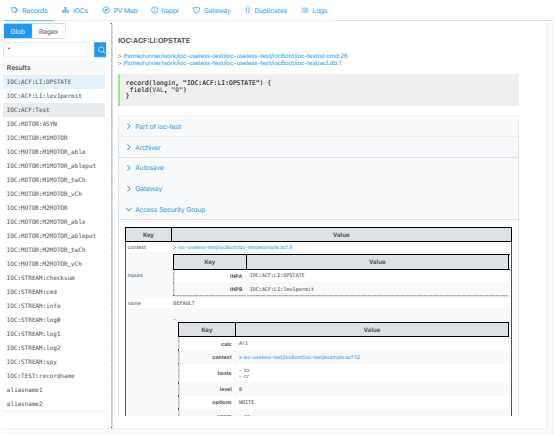

Figure 3: whatrecord frontend: a record with access security group settings.

whatrecord Lint This command offers a work-in-progress set of linting tools, most of which has not yet been well-defined. The goal for these tools will be to allow a facility to enforce certain standards for their IOC files and avoid common pitfalls by detecting issues prior to the IOC boot process.

WEB FRONTEND

A Vue.js [12] web-based frontend application is packaged separately in the `whatrecord` repository. It provides a user-friendly view of the information that `whatrecord` can parse and aggregate.

A searchable index of records is the primary view. Record information can be used to correlate back to other tools and views. For example, when an autosave configuration is found in an IOC, the frontend will display that information in a table underneath "Autosave" and annotate the record to show the on-restore value.

Similarly, StreamDevice protocol files, happi devices, asyn and motor port, Archiver Appliance, access security groups, and others will be displayed alongside the record information. Source code files may be viewed by clicking on any context link.

Record links, when detected, are displayed as an interactive graph (by way of `d3-graphviz` [13]). Inter-IOC links can be interactively investigated in the "PV Map" section.

Sample screenshots of the frontend are shown in Figs. 3 and 4.

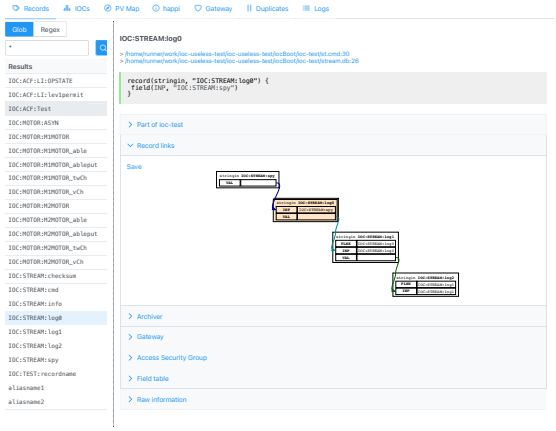

Figure 4: whatrecord frontend: a record with links.

LCLS

Views for LCLS-Specific tools can be enabled with an environment variable setting, including a view of happi devices, an LDAP / netconfig settings viewer, and epicsArch PV listings.

GitHub Actions and "Offline" Mode

A sample repository [14] is provided to exhibit the "offline" mode supported by `whatrecord`.

The offline mode utilizes the `whatrecord` server in continuous integration to generate a snapshot of the server information. This information then takes the place of a backend server, requiring only a tarball of JSON in order to populate the data.

Trying whatrecord

The easiest method to try the frontend alongside the backend is with Docker Compose.

```
$ git clone https://github.com/pcdshub/whatrecord
$ cd whatrecord/docker
$ docker-compose up
```

After executing the above, wait for a few minutes and then open `http://localhost:8896` in a browser.

The parsing tools can be used directly with a working Python 3.9+ environment:

```
$ pip install whatrecord
$ whatrecord --help
```

The `whatrecord` source code is available on GitHub [1] and documentation is available on GitHub Pages [15].

REFERENCES

[1] whatrecord: source code repository, `http://www.github.com/pcdshub/whatrecord`

[2] EPICS, `http://www.aps.anl.gov/epics/`

[3] StreamDevice support, https://paulscherrerinstitute.github.io/StreamDevice/index.html

[4] Lark: a parsing toolkit for Python, https://github.com/lark-parser/lark/

[5] sumo: EPICS support module manager, https://epics-sumo.sourceforge.io/

[6] epicsmacrolib: EPICS-compliant macro handling, https://github.com/pcdshub/epicsmacrolib

[7] pytmc: TwinCAT PLC Code to EPICS Database Tool, https://github.com/pcdshub/pytmc/

[8] jq: JSON filter command-line tool, https://jqlang.github.io/jq/manual/

[9] Bluesky, https://blueskyproject.io/

[10] happi: the LCLS ophyd device database, https://github.com/pcdshub/happi/

[11] aiohttp: an asyncio HTTP client/server, https://docs.aiohttp.org/en/stable/

[12] Vue.js, https://vuejs.org/

[13] d3-graphviz, https://github.com/magjac/d3-graphviz

[14] GitHub Actions and whatrecord sample, https://github.com/pcdshub/ioc-whatrecord-example

[15] whatrecord: documentation, http://pcdshub.github.io/whatrecord