

C³PO

Bernhard W. Adams, June 27, 2019

I. INTRODUCTION

C³PO (Control Code Collection for Process Organization) is a set of inter-process communication standards for interfacing industrial-control devices to users. It is implemented as a collection of programs, tied together by a PostgreSQL database. This is a highly modular approach; each of the programs can be written in just about any programming language (python being used at this time), and a good part of the functionality of C³PO is not found anywhere in program code, but rather in the database entries (defined through setup files) that tie the programs together. Component programs interface with each other through operating-system calls or through TCP sockets using secure-socket-layer communication. Multiple users can access C³PO at the same time, with conflicting access being regulated through database locks. Furthermore, C³PO can be distributed among several computers, as long as they are on a network to allow TCP/IP communication amongst them.

The database need not be accessed directly by the end user, but it may be helpful to peek into it for debugging purposes. C³PO isolates the end user(s) from the gory details of bit-flipping, and other highly hardware-specific tasks. These are outsourced to driver routines that need to be supplied for each hardware device used in an experiment. The end user(s) can then concentrate on the real tasks of moving actuators, setting temperatures, reading sensors, etc.

Experiment control can be abstracted as 1) converting real-world items of interest to data (numeric, string, image, etc.), 2) processing the data in a program, and 3) converting data to a real-world effect. A good approach is to discriminate between hardware-abstraction layers where the lowest ones provide direct interface with the hardware, intermediate ones handle standard tasks, such as scanning, and the uppermost ones are the user interface, which presents itself to the experiment in terms of functionality, as opposed to specific devices. For example, the end user wants to move something by some amount of millimeters without regard to the specific vendor, or wants to measure a voltage with whichever instrument is on the experiment. C³PO presents the instrumentation in this functional form.

On top of that, the experimenter can build scripts that control the experiment, or make use of built-in standard features, such as multi-dimensional scans. The interface between the low-level drivers and the lowest hardware-abstracted software needs to be standardized, yet flexible. In C³PO, the real world is abstracted through process variables (PVs). Data formats of PVs can be numeric, string, or vectors or matrices of these. Figure 1 shows a schematic representation of the role of C³PO code in an experiment providing multiple users with access to multiple instruments to control, measure, perform scans, etc.

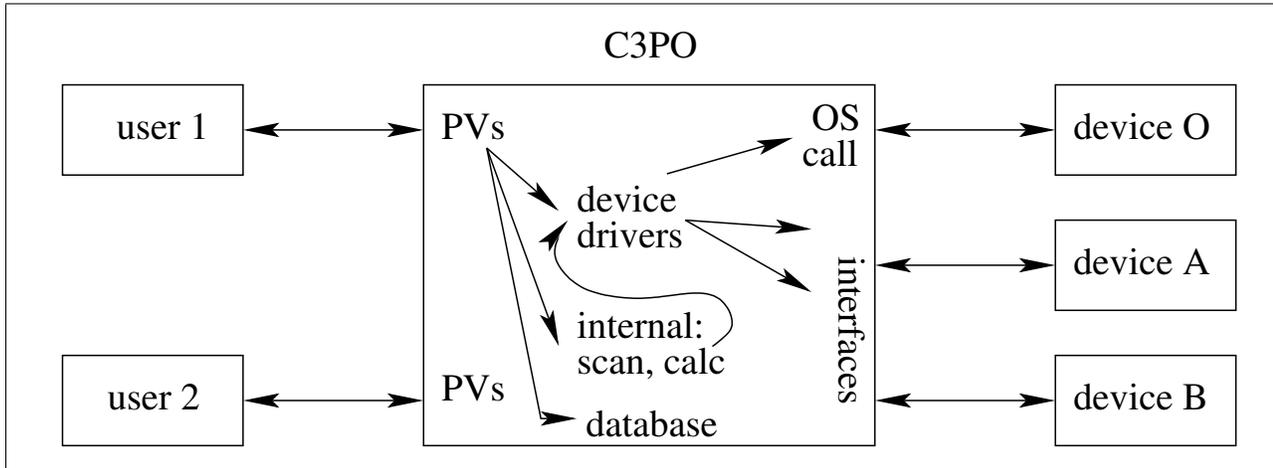


FIG. 1: Schematic representation of the role of C³PO code as an interpreter between user(s) and devices in an experimental setup. Users write to and read from PVs. These are resolved internally in C³PO to call up appropriate device drivers or C³PO-internal functions (scans, user calculations, etc.), and/or access database entries. Device drivers communicate with instruments (actuators, measurement devices, etc.) through OS calls or interface drivers. See text for more detail.

On the right-hand side in the figure are several devices, which might be voltmeters, power supplies, actuator controllers, oscilloscopes, etc. Each device has its own command protocol and one or more communication interfaces. To the user(s), the devices are abstracted as PVs to write to (ex. actuator position) or read from (ex. voltage). The C³PO code contains drivers that translate PVs into commands for the devices, as well as interface drivers that route these commands and the replies through specific interfaces (serial ports, TCP/IP, USB, etc.). For reconfiguring the interfaces through which a device communicates with the control computer some database entries need to be modified, but the user need not be aware of any such change.

When a PV is used (written to or read from), the lowest-level HA program looks up the PV name in the database to find the following information:

- name of the driver program, as known to the OS command line
- type: read/write
- optional side effect: the name of an OS-command-line program that may do other things. This may sound rather harmless at first, but it may be used to ensure database consistency (see below), or for advanced operations, such as sending an email message if a motor is moved, or measured value exceeds a threshold. Side effects are handled by the PV gateway program pva.py

As far as the end user is concerned, there are only two types of PVs: those that one writes to and those that one reads from. Internally, i.e., not visible to the end user, there are more types, which are discussed in Sec. II. A PV name can be any string not starting with the asterisk (see below). Typically, PV names are structured as <instrument name>.<function>, for example `k2410.rcurr` to read a current from a Keithley 2410 picoammeter. However, that is only a convention used in the current implementation, and not a necessity.

C³PO will store in its database the last value read from a read-type PV and the last value written to a write-type one. Reading from a PV with an asterisk pre-pended to its name will not actually access the respective instrument, but will rather return that last-read/written value. This is the only case where a write-type PV can be read from. In all other cases read and write types are distinct from each other, even if they may have the same name (this may seem to be a semantic distinction here, but it will become important later on).

Drivers are command-line executable programs. However, users should access drivers only through the HA layer structure of C³PO, and never directly. A driver program must accept at least one argument that tells it what to do, as well as additional optional arguments that depend on the requested function. All drivers must support the functions listed in Table. ??.

The approach of casting an experiment into a collection of PVs is intentionally constraining, so that a clear structure is available. One might think of the PV structure as the skeleton of the experiment. There are, however, many situations where a more free-style scripting approach is convenient. There are several places in the PV structure where this is possible, and this might be called putting flesh on the skeleton. At the highest HA levels, a user will simply write python scripts that do calculations and other complex operations before even calling a core C³PO script to use the PV structure. There is also a way of calling a python script, or any other program that can be started from the command line: The standard installation includes a scripting PV and a driver associated with it. More scripting PVs using that driver can be defined by the user. When a value is written to a scripting PV, then a command-line program is called with parameters that can be defined as described in Sec. II, and a return value is also provided through a PV.

C³PO can run in one of three modes:

- normal: actuators move, detectors are read, etc.
- read-only: no actuators move, but detectors are triggered and are read after the prescribed wait times
- full-simulation: no actuators move, and no detectors are triggered or read

Data files will be written to in all three modes.

Several users may simultaneously access C³PO, and multiple experiments can be controlled simultaneously by one installation of C³PO. Normally, users would write to (directly or within a scan) disjunct sets of devices. However, for cases where two users may want to write to the same PV, a database-lock mechanism prevents confusion. The way that this works is that each time a PV is to be written to, C³PO first ‘checks it out’ of the database and puts a lock on it, which it returns only after the transaction is completed. If the PV is already locked, C³PO will wait. These locks, henceforth called “C³PO lock” are database entries and are quite distinct from the locks that the database itself uses to synchronize transactions. In the case of a program crash, locks may be left un-returned, i.e. the database would still contain an entry for a checked-out lock. In order to clear such “stale locks”, one may issue an explicit command to clear a lock on a specific PV, or all of them.

II. USAGE

At this time, C3PO is mainly accessed through the command line of the OS (preferably Linux), but a graphical user interface (GUI) is under development.

A. Command-Line Interface

The two most important functions on the command-line interface are `wpv` (write to a PV) and `rpv` (read from a PV). They are currently implemented in python, but may be written in some other language, as well. These functions are found in the “user” subdirectory of the C3PO installation path. To write to a PV (here using the python version on a UNIX system), type `./wpv.py <PV name> <value>`, for example `./wpv.py xtr.mvabs 10` to move a motor “xtr” to an absolute position of 10 units (mm or whatever). NB: The base name “xtr” of the PV, its extension “mvabs”, or even the structure of abse and extension, are all site dependent, and so is the choice of mm as units. To read from a PV, type `./rpv.py <PV name>`. Here are some typical examples:

command	meaning	comment
<code>./wpv.py xtr.mvabs 10</code>	move xtr to the position of 10	
<code>./rpv.py xtr.mposn</code>	read current position of actuator xtr	
<code>./rpv.py Xtr.mposn</code>	read current position of actuator Xtr	<i>not the same as xtr</i>
<code>./wpv.py scanD2.Trig 1</code>	trigger scan “D2”	outer loop of a 2d scan, see Sec. II C

TABLE I:

B. GUI

C. Scans

Scanning, i.e., systematically changing one or more actuator positions, or voltages, temperature, etc., and taking measurements at each point, is one of the most important functions of an experiment-control program. Colloquially, “scan” may refer to both the actual process of scan execution, but also to the type of scan (linear vs. table scan, or which dimension in a multi-dimensional scan). In cases below where there may be doubt. the usage will be clarified.

In C3PO, scan parameters (which actuators to move and how which detectors to read, etc.) are contained in the master database, from which the scan driver reads them during scan execution. Values for these scan parameters are sent to the database by writing to pertinent PVs. This is usually handled by editing a text file, and then calling a program to handle all the writing of PVs. The set of all PVs associated with a scan is called the scan-PV set. Several such sets are pre-defined in C3PO in such a way that the associations between them are already defined.[1] Once all scan parameters are defined, the scan is started by writing a value of 1 to a scan-trigger PV, such as in `./wpv.py scanD2.Trig 1`.

Each of the scan-PV sets has the same kind of associations within it, and there are none cross-linking the sets. In particular, there is nothing in C3PO itself that would designate a scan-PV set as representing a particular dimension for a multidimensional scan. Rather, multi-dimensional scans, even those of non-integer dimensionality, are set-up by the user by writing a scan-trigger PV to a place in the scan-definition file (the contents of which go to respective PVs in the scan-PV set) where-ever a write-type PV can go, such as places for actuators, detector triggers, etc. This is similar to LEGO bricks in that a set of scan PVs is analogous to a brick, and bricks can be assembled in all kinds of multi-dimensional structures.

Two types of scans are pre-defined through their sets of scan PVs: linear scans and table scans. In a linear scan, one or more actuators are moved (or set in the case of a voltage, or temperature, etc.) in a sequence of steps, and zero or more detectors are read in each of them. The steps are given as a linear progression of parameter values between two endpoints. This may be as simple as stepping through equidistant positional values of an actuator, or more subtle such as stepping through equidistant angles, which translate non-linearly into actuator positions. In any case, a linear scan consists of a progression through a given number of steps, where something happens in each step. If that something contains the triggering of another scan[2], then one gets a multi-dimensional scan. Additionally, there are a “before” and an “after” section in a linear scan, which serve for initialization and

“mop-up”. Other scans may also be triggered in those sections. There is a lot more detail on linear scans in Sec. II.

Another type of scan is the “table scan” where parameter values for actuators are not stepped linearly, but are rather read from a file. Actually, the linear scan internally generates a table (and exports it to a file for debugging purposes). Thus, internally, a scan is always run as a table scan.

Finally, even though technically not a scan, a C3PO script can also be used to step through a sequence of positions, etc., and take measurements in each.

1. Linear Scans

A linear scan is one where one or more parameters are stepped linearly through a succession of values, which become, in some linear or non-linear way, actuator values.

extension	meaning
Before	;-separated list of wPVs and values or variable assignments to initialize scan
TimeInterval	minimum time to wait after completion of move until next
Actuators	;-separated list of write PVs: actuator names
ActWait	;-separated list of read-type PVs to ask if actuators are done moving; same length as Actuators
ActSettle	list of actuator settling times
ActFrom	
ActTo	
Npoints	one integer N : number of scan points; $I = 0 \dots N - 1$
Safety	list of ;-separated expressions returning bool: proceed only if all are true
MeasurePrep	
DetTrg	
DetTrv	???
DetWait	
DetRdTrg	
DetRead	
AbortList	
After	

TABLE II: Summary of scan-setup PVs formed by taking the scan base name and appending the extension in the first column, above, i.e., if scan is the basename, the first row would expand to the PV scan.Before

In the Before section, variables with names \$A .. \$Z can be assigned values from rPVs

Here is some more detail on the setup PVs. In order to define the exact syntax, the regex used by the scan driver Scan.py is given in some cases. The semicolon is used to separate list items using the regex `\s*;\s+`

- Before: a list of semicolon-separated items. Each item can be either a space-separated pair of `<wPV name> <value>`, or a set of (`<variable name> <assignment operator <-> <rPV>`) recognized by the regex `^\s*\$[A-Z]\s*<-\s*\$+.*$,` where \$A .. \$Z are possible variable names, except reserved names \$I and \$L. The purpose of the latter is to make the scan parametrically dependent on the reading of a PV. This may be some outcome of a measurement in the experimental setup, or a RW PV by which another part of C3PO can broadcast information. For example, an outer-loop scan can transmit its scan index to an inner-loop scan using this RW-PV mechanism. The reserved variables are assigned values in each scan point: \$I= 0 .. N - 1 is the index of the scan point where N is given in Npoints, and \$L is, depending on which ;-separated column it is in, the interpolated position between the respective actuator’s from and to values
- Actuators: a list of semicolon-separated items that are either wPVs or space separated pairs of a wPV and an expression that can be evaluated to a number or a string.
- ActWait: a list of semicolon-separated rPVs. The scan will wait after moving actuators until all these rPVs give a non-zero reading. Empty list items or ones commented out with a pair of parentheses will be ignored

III. STRUCTURE

C³PO is structured in hardware-abstraction (HA) layers, with lower-level, system-internal and hardware-specific (HA:-1 and lower), and higher-level user (HA:1 and higher) routines. Between these two sets (HA:> 0 and HA:< 0) lies HA:0 represented by a single gateway routine, `pva.py` (PV access), through which all communication between the user and C³PO goes. Similarly, all communication between C³PO and the devices passes through a single interface-gateway routine called `intAcc`, which handles selection of the proper interface for a requested device. HA:< 0 routines are not accessed directly by the user, and no knowledge of them is required to use C³PO. The parts of C³PO are tied together by a postgresql database, which is set-up individually for a given experimental setup (in a largely automatic process described in Sec. II). The structure of C³PO is shown graphically in Fig. 2. All the end user needs to know is then how to interface with the `pva` gateway, and all the person setting up the experiment (the “installer”) needs to know is how to specify interfaces for the `intAcc` gateway. All other parts of C³PO need to be understood only by code developers. The figure shows how communication between i) a multiplicity of user

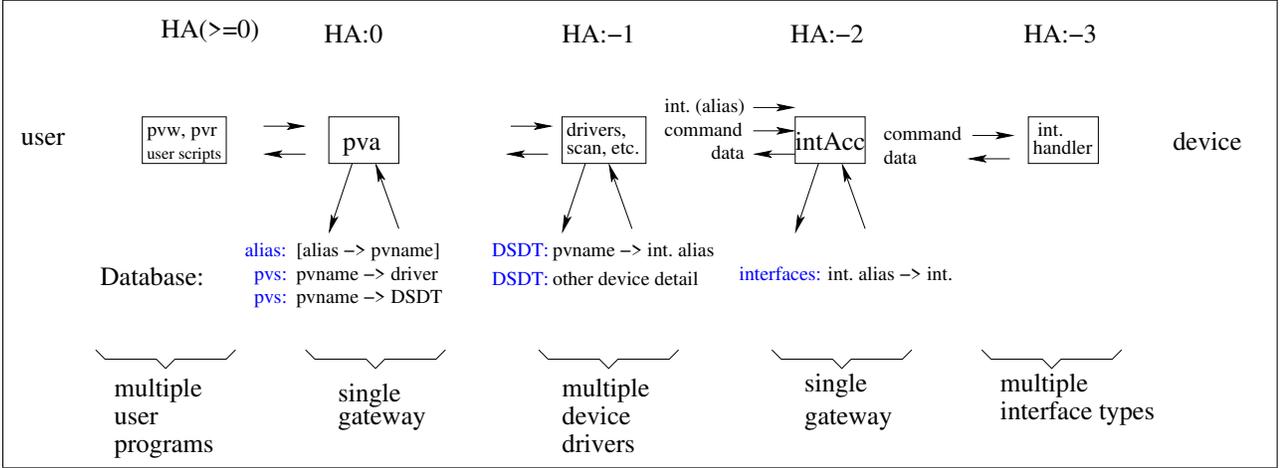


FIG. 2: Graphical representation of the structure of C³PO.

programs and ii) a multiplicity of devices connected to the computer by iii) a multiplicity of interface types is channeled through two gateways where the flow of information is guided by entries in the database. This makes it easy to set-up an experiment: provided the device drivers and interface handlers exist (as code snippets), then the person setting up an experiment only needs to generate database entries that associate PVs (i.e., abstract representations of functionality like “read a current”) with device drivers and that associate devices with interfaces that they are connected to.

This will now be explained in detail: A user-issued command to write to or read from a PV ends up in a call to `pva`, then `pva` looks into a database table called `pvs` to find a driver program for the corresponding device (physical or virtual) and a device-specific database table (DSDT) with further information. It then calls the driver and hands to it the PV name and the DSDT, which contains specific information on the action to take with a given PV, and which interface to use (by a symbolic interface name). The driver then generates command strings, etc. appropriate for the action to be taken (for example a string instructing a voltmeter to take a reading and return the result), and passes the command and the symbolic interface name to the interface gateway `intAcc`. `intAcc` then consults another database table to resolve the symbolic interface name into a real one, and to modify the command string accordingly (for example, embed the command in a string containing additional information, such as GPIB bus address). Thus, if a device is connected to a different interface, or its sub-address on a bus (GPIB, modbus, etc.) changes, then the corresponding entry in the `interfaces` database table needs to be modified. Conversely, a reply by a device is modified in `intAcc` according to interface-specific rules specified in the `interfaces` table (for example echo stripped off), and then passed to the driver, which interprets it to produce a reply to the user (in case of a PV read operation).

It is important to know where changes in C³PO must be made when the experimental setup changes. For every addition or removal of an instrument, or every change of the way an instrument is connected, a change must be made in exactly one place in the database structure, and for new instrument type, exactly one driver program must be supplied. Because it is so important, it will be repeated once more: Every change in the experiment’s configuration requires changes in exactly one file from which database entries are generated.

- when a new device is introduced into a setup, then a file defining entries in a new database table for this devices must be generated (or parameters in an existing definition file must be changed to generate such a table - see Sec. II). Furthermore, it may be necessary to write a device driver if it doesn't yet exist for this type of instrument. The driver needs to generate command for the instrument and interpret replies.
- when the interface assigned to a device is changed, then the entry in the interfaces database table need to be changed. This applies for simple changes, such as using a different serial port, or a different GPIB address or a different TCP address to more drastic ones, such as using an ethernet interface for an oscilloscope instead of its USB interface.

If the connections of physical devices to physical interfaces changes, such as a swap or re-assignment of serial ports, then this change needs to be entered into the `interfaces` table, but everything else stays the same. This is even the case if the type of interface changes, such as when an oscilloscope is no longer connected through a USB interface, but rather through ethernet - all that is necessary is to update the entry in `interfaces`. `intAcc` can communicate with interface handlers by calling them through the operating system, or (preferred) by sending a message through a TCP socket to a daemon interface handler.

Because PV names can get somewhat lengthy, C³PO provides the means to use alias (shorthand) names for them. These are stored in a table called `alias` (see Sec. II for details). Alias names are resolved at the HA(-3) level and are context-specific (see below). All other tables are not relevant to the end user. For a complete overview of database tables, see Sec. II.

The exact nature and number of HA:1 routines present in a site installation depends on the experiment, but it should be kept to a minimum. Typical HA:1 routines would be to move a motor, read a detector, etc. Even if, for example, different types of motors are combined that require different hardware drivers, a single HA:1 routine is sufficient because all it does is to write the destination to a PV. Resolution of drivers, etc. is done by `pvr.py` when it looks up the PV in the database. Therefore, the HA:1 routines are, typically, representative of elementary tasks like “move something”, “read something”, etc. If hardware is changed, for example one type of motor replaces another, the user will, ideally, not even notice the change. Although possible, in principle, to call the HA:0 routines from HA-2 and higher, it is good style to *not* do so, but rather always go through HA:1. Discipline in that respect will be very helpful in debugging an experiment.

The files that comprise a C³PO installation reside in several directories, which are, here, all referenced relative to the C³PO root directory. End users should only access routines in the `./user` directory and, if necessary, write python routines for experiment automation in that directory. Data are stored by default in the `./DataFiles` directory, and logs are kept in the `./logs` directory. In the process of a C³PO site installation, the contents of the `./user` directory (but not `./DataFiles` or `./logs`) are deleted, and are replaced with the installation default. Users should therefore back-up their own scripts somewhere else, so they are not lost when the site installation is re-done or updated. If a routine is used often in an experiment, then it can be saved in the `./CustomFilesXXX/user` directory from whence it is copied into `./user` by the installation routine (where XXX stands for a string specific to the installation), see Sec. II for details on the installation process. The directory `./docu` contains documentation. The `./C3PO` directory contains installation-specific routines (drivers, etc.) that are created by the person installing C³PO. Its contents are also overwritten in the installation process by files stored on `./CustomFilesXXX/C3PO`, modified by the installation program using information stored in the `./setup` file. Finally, the directory `./installation` contains core code only. There is nothing in there that needs to be modified by the installer, and end users should not even entertain the thought of going there (I see, you already secretly took a peek).

The simplest use of PVs is for single effects, i.e, writing to a PV to, say, move a motor, or read a detector through a PV. Some other common scenarios are implemented in C³PO, namely user calculations and scans. A user calculation is a routine that reads from PVs, performs a calculation on them, and returns the result as a PV. User calcs are started and stopped by writing 1 or 0 to associated trigger PVs. A scan is a stepwise change of one or more PVs, and acquisition of data in each step. Scans may be multi-dimensional, i.e., the data acquisition of an outer scan consists of triggering an inner scan.

A. How the PV Structure Works

In order to properly use C³PO, it is very helpful to understand the relationship between PVs and real-world events. It is also helpful to understand that user-accessible PVs are always represented in two (or in rare cases more than 2) database tables, namely the main `pvs` table, and in at least one (rarely more than that) driver-specific table. With this split, all PVs present themselves to the user in the exact same way, the only distinction being whether a PV is of type ‘W’ (write-to) or ‘R’ (read-from). The entries in the `pvs` table contain pointers to the driver and to another, driver-specific database table that C³PO is to use to work with that PV. Frequently in this

document, there will also be mention of other PV types, such as ‘T’ for ‘trigger’, ‘S’ for ‘setup’, etc., but these are simply write-type PVs with a particular function. Trigger Pvs are used to start something that has been set up before by writing to the proper setup PVs, such as scans, timers, user calculations, etc. Most often, writing a ‘1’ to a trigger PV will start the associated process, and a ‘0’ will stop it before it would do so by itself, but there are other possible trigger values, too, depending on the function to be triggered.

From the user perspective, the most basic operations that everything else (scripts to control an experiment, etc.) is built on, are operating-system (OS) calls to `pvr.py <callfrom> <PV name>` to read from a PV, or OS calls to `pvw.py <callfrom> <PV name> <value>` to write a value to a PV. Here, the command-line argument `callfrom` is the name of the script from whence the call comes, `PV name` is the name of the PV (or an alias name) to write to or read from, and `value` is the string or numerical value to be written. The `callfrom` information is mainly required for alias resolution (see below). Although, in principle, an end user can directly call `pvr` or `pvw` from the command line, this is not advisable. Rather, one should always go through function-specific (move motor, read detector, etc.) HA-1 scripts that do the calls to `pvr` and `pvw`.

`Pvr` and `pvw` look first into the `alias` database table to see if the PV name passed to them is an alias name, and, if so, resolve it to the actual PV name. If none is found, it is assumed that the name is an actual PV name. Alias-name resolution depends on the context, i.e., the `callfrom` command-line parameter. The same alias name may thus mean entirely different things if used by different HA-1 routines. This is a very practical feature because different functions of the same device (such as motor move, motor readback, etc.) are performed through distinct PVs, but these can all have the same alias name.

With, now, the actual PV name in hand (directly or after alias resolution), `pvr.py` and `pvw.py` look into the `pvs` table to retrieve from it the entry with the PV name, which contains further information, such as the name of the driver routine, name of another database table specific to the driver, etc. `Pvr` and `pvw` then call the driver whose file name they found in the `pvs` table, and give exactly the following information to it as command-line parameters of the operating system:

- the keyword “EXEC” (drivers can perform other tasks described in Sec. II),
- the `callfrom` parameter and its own file name (`pvr.py` or `pvw.py`),
- the name of the driver-specific table found from the entry in `pvs`,
- and the value of the PV to be written (in the case of `pvw.py`).

The driver-specific database table contains corresponding entries for the PVs, i.e., every PV in the `pvs` table needs to have a corresponding entry with the same name in some driver-specific table. However, the driver-specific tables may contain further PVs that are not in the `pvs` table, but are rather meant for internal use only. The entries in the driver-specific tables contain further information for the driver, such as physical interface (serial port, etc.), or other information that the end user does not need to worry about.

Depending on the driver, a good part of its functionality may be contained in the relations between PVs in the driver-specific database table. PV entries in a driver-specific may be of the following types: ‘R’ for ‘read’, ‘W’ for ‘write’, ‘I’ for internal use, ‘S’ for setup, or ‘A(..)’ for ‘arithmetic’ specified by the contents of the parentheses. A W-type PV in the `pvs` table may correspond to a W-type or S-type PV in the specific table, and an R-type PV in `pvs` may correspond to an R-type, or an A-type PV in the specific table. I-type PVs are not user-accessible, except by peeking directly into `psql`. Good examples of this are the scan (see Sec. II) and software-timer drivers (see Sec. II).

B. Setup, trigger, and status PVs

Even though setup and trigger PVs are nothing but write-type PVs, and status PVs are simply read-type ones, it is helpful to take a closer look at the roles they play in C³PO: Often times, a real-world effect can be achieved by simply writing a value to a PV, for example send an actuator to some position. However, there are other, more complex cases, such as running a scan. Before a scan can run, one has to set it up by specifying what to move, what to read back, where to write the data, etc. Such cases are handled in C³PO in the following way: In the pertinent driver-specific database table, i.e., scans as part of the standard installation, there are several PVs of type ‘S’ for setup with corresponding entries of type ‘W’ in the `pvs` table. When a value V is written to such a PV, `pvw.py` finds the entry in the `pvs` table, retrieves the names of the driver and the driver-specific database table, and calls the driver with that information. The driver will then find the PV in its database table, and, because it is of type ‘S’, will enter the value V into a field in the database table called ‘ivalue’. Nothing more happens at this point.

There is also at least one PV entry of type ‘T’ for ‘trigger’ in the driver-specific database table. Its corresponding entry in the `pvs` table is of type ‘W’. There are several columns in the driver-specific table, which are populated only in a row for a trigger PV. These entries give the names of the setup PVs that hold the information needed to run the scan, etc. When a value T is written to such a PV, `pvw.py` calls the driver, as described above. The driver then finds the names of the setup PVs, retrieves the setup information from the ‘`ivalue`’ field in each of them, and then proceeds with the operation. There may be more than one trigger PV, typically related to different setup PVs. For example, there is a dedicated trigger PV for each dimension in a multidimensional scan. Each of these will be related to a different set of setup PVs to hold the information which actuators to move in that particular dimension of the scan, etc. Some of the setup PVs may, however, be the same for different trigger PVs, such as the one that holds the name of the data file in a multidimensional scan in case all scan data should be collected in one file.

C. Specifying PVs for an Experiment

A part of setting up an experiment is to specify the PVs for all devices. Each type of device (pA meter, HV supply, etc.) has its own device-specific database table (DSDT), the contents of which are defined in a file with a name that ends in ‘`_Defs`’. The installation scripts search for all such files, generate the DSDTs, and automatically enter all the PVs also in a database table named PVs, which is the go-to location for the user-interface gateway program `pva` to find out what to do with a given PV (see Sec. II). Two technical notes: Defs files are executable, which have in the first line a “shebang” that directs further processing to the program `setupDatabase.py`. The setup scripts automatically call a program `generatePVlist.py` that generates an file `Pvs_Defs_user` that is then processed in the same way as the other Defs files. Entries in a DSDT specification file may be parametrized in several ways, as described in the following subsections:

1. *foreach / unforeach*

The `foreach` statement allows the generation of PVs for multiple devices of the same type from a single entry in the DSDT file. Its syntax is: `# foreach key in {comma-separated list}` in a line by itself (the regex pattern is `^\s*#[#\s]*foreach\s+\S+\s+in\s+\{.+?\}\s*$`). Every line in the Defs file that contains the key in the first column (the one for the PV name) will then generate multiple entries in the database, one for each item in the list inserted for the key. Corresponding replacements are then done in other columns, as well, but the `foreach` replacement is triggered for an entry in the Defs file only if the key is in the first column. This mechanism is effective only for lines in the Defs file following the `foreach` statement. A `foreach` statement can also be cancelled, so it is not effective for any lines following one with the syntax `# unforeach key` (the regex is `#[#\s]*unforeach\s+\S+`).

For example, a Defs file might contain the following lines:

```
\#foreach_S_in{a,b}
HV_S_.id,R,*IDN,,HV_S_
```

This will then generate two PVs named `HVa.idn` and `HVb.idn` (to obtain the ID reply from a Stanford PS350 high-voltage supply in this case).

Multiple `foreach` replacements may be present in a single line in the Defs file. This will then generate a number of PVs given by the product of the numbers of items in each of the `foreach`-item lists. An alternate mechanism with very similar effect is described below under “Indices”. Whether to use `foreach` or `indices` is only a matter of convenience.

2. *define / undef*

Is applied before `foreach` expansion and works as in C

3. *replace / unrep*

Is applied after `foreach` expansion and works otherwise like `define`

4. [] Indices

There are two types of indices, those in square brackets and those in curly brackets. The latter will be described below. Indices in square brackets have the same effect as a foreach statement, but only for the line in which they occur. Square-bracketed indices can be defined as lists or as ranges.

A range is given by numbers or letters a, b or a, b, i in the form of $[a : b]$ or $[i = a : b]$, where a is the first index, b is one beyond the last one (in the python way), and the optional i is a range identifier that may become necessary in case there are multiple ranges in a PV name (see below). If a and b are numbers, the indices include $a \dots b - 1$, otherwise indices run from alphanumerical ASCII characters for a to the one before b . The following points should be noted:

- a must be smaller than b , either numerically, or in the sequence of ASCII characters, i.e., 0 (0x30) is smaller than A (0x41), and A (0x41) is smaller than a (0x61), etc.
- if a is a number, and b is not, then a needs to be single-digit (because it is being interpreted as an ASCII character)
- only the alphanumerical ASCII characters in a given range are used, ASCII-wise, there would be several special characters between 9 (0x39) and A (0x41), but these are being skipped, and likewise for those between Z (0x5A) and a (0x61)

The setup script `setupDatabase.py` will internally convert a range into a list, and then process it.

A list is given in the form of a comma-separated sequence of items in the form $[a, b, c, \dots]$ or $[i = a, b, c, \dots]$. As mentioned above, and described in more detail below, the optional identifier i serves to disambiguate among multiple lists. When generating database entries, `setupDatabase.py` will, for each entry in the list, replace the square brackets and its contents with one item in the list (whether specified as such, or generated from a range). It will do so for lists in all columns of a line in the Defs file, but only if that list does appear in the first column, i.e., the one for the PV name.

There may be multiple ranges or lists in one PV name. As long as the lists are substantially distinct, the properly corresponding ones in other columns will be identified correctly and will be replaced by the pertinent list items.

5. {} Indices

The other type of range or list is enclosed in curly brackets. This one will not lead to separate PV entries for each of its items, but rather a comma-separated list in place, where each item in that list has the curly brackets and its contents replaced with one item from the list in curly brackets. Unlike with square brackets, this will occur independently of whether the curly brackets appear in the PV-name column. The rules for converting ranges to lists (ASCII and such) apply in the same way as for square brackets.

6. Example

An example of a DSDT specification file is shown in Fig. II.

FIG. 3:

IV. FLOW

Access to a PV, whether directly through `rpv.py` or `wpv.py`, or indirectly, for example from within a scan, occurs by the following sequence of events:

- the program that needs to access a PV (`rpv/wpv` or something internal to C³PO) calls the central PV gateway `pva.py` and gives it the PV name or alias (see Sec. II), and, in case of a PV-write operation, also the value to be written

- `pva.py` resolves PV alias names (if necessary), then looks up the PV in the `pvs` database table to find the device driver and device-specific database, then calls the device driver (internal programs scan as the scan driver will receive information from `pva` to call the driver directly, thus bypassing some communications overhead)
- the device driver looks up further information for the PV in the device-specific database table, including, in particular, the interface name or alias name (not to be confused with a PV alias), generates the command string, then calls the interface gateway `intAcc` and passes the command string and the interface name/alias to it
- `intAcc` looks up the interface name/alias in the `interfaces` database table. Actual physical interfaces are either represented by OS (name beginning with `OS_`) calls or SSL TCP sockets (name beginning with `SSL`). Interface aliases, protocol wrappers, etc. have names beginning with anything but `OS_` or `SSL` or the special character `&` which is reserved for internal use (see below). If `intAcc` receives an `OS` or `SSL` interface name from the device driver, it will call that one directly. Otherwise, it will resolve what may be an extended chain of alias names and protocol wrappers. It is good practice to never specify a physical interface (`OS` or `SSL`) directly in the DSDT definition file, but always refer to an alias. This practice facilitates any modifications in device interfaces, such as when a device is moved from a serial to GPIB interface because then one needs to make changes in the `interfaces` database table instead of searching through DSDT definition files. Aliases and protocol wrappers may be chained. For example, an alias name for a device, say “iK6485” for the interface to which a certain picoammeter is connected might resolve to a sequence “NIsGPIB / SSLs232_10” meaning that the picoammeter is connected to a GPIB bus, which is connected to the host computer through a serial-to-GPIB converter. Then, the entry for K6485 would contain the GPIB address, and the entry for NIsGPIB would contain protocol-wrapping information (how to tell the serial-GPIB converter the GPIB address, how much to write or read, etc.), and the entry for SSLs232_10 would contain serial-specific information, such as baud rate, etc. Other GPIB instruments could also refer to the same chain of NIsGPIB / SSLs232_10 without duplicating the information specific to the serial-GPIB converter or the serial link. In order to minimize database traffic, `intAcc` will automatically generate database entries for fully resolved chains whenever a chain is accessed for the first time. These entries have an ampersand character prepended to the interface alias name.

A. The interfaces DB Table

The `interfaces` database table defines all the interfaces through which C³PO communicates with the devices in an experiment. It is, functionally, a mirror image of the `pvs` database table that contains entries for all the PVs that a user can access. The `interfaces` table can contain entries for actual physical interfaces, alias names for them, or protocol-wrapper alias names. Interface names or interface aliases are for use by device drivers, i.e., they can be Protocol wrappers are to be used only by `intAcc`. Examples are given in Tab. III and are described in Sec.II.

The columns in this table correspond to columns in the database:

- **id:** internal for `psql`
- **name:** name of the interface, interface alias, or protocol-wrapper alias
- **dnPort:** if **name** is:
 - a physical interface, the port in the computer corresponding to the interface, for example, `/dev/ttyUSB0` for a USB-serial port
 - an interface alias, the name of the actual interface, for example, if K6485a is the name of an interface alias, then `SSLs232_1` might be the actual interface represented by the alias. The latter needs to be represented in the `interfaces` DB table, so `intAcc` can find it there.
 - a protocol-wrapper alias,
- **upPort:** in the case of
 - a physical interface, where the interface driver communicates through TCP sockets (interface names beginning with `SSL` or `TCP`), the host name (local or remote) and the TCP port to access the interface driver

– a physical interface, where the driver is called through the OS (interface names beginning with OS_, rare, not recommended)

–
–
–

- **if**
- **if**
- **if**
- **if**
- **comment**

The rows in the DB table can contain:

- a physical interface
- an interface alias

id	name	dnPort	upPort	driver	start	call	daemon	comment
1	SSLs232.1	/dev/ttyUSB0	localhost 12347	SSLs232.py	9600 8 N 1 xoff 1 0.1			# comment

TABLE III: Some sample entries in the `interfaces` DB table

B. Interface Drivers

Between C³PO and the devices controlled by it are computer interfaces that are managed through the `intAcc` component of C³PO (see Fig. II).

The last step in communicating with devices controlled by C³PO is

Interface drives are background processes that handle the communication with particular interfaces (serial ports, etc.). An interface is started with one parameter that tells it which port to listen to.

V. USER CALCS

User calculations are useful for providing values that are computed as some expression of PVs. Several user calcs UC1.4 are provided as part of the standard installation. They are set up in the usual way of links between status, setup, and trigger PVs. However, unlike other driver scripts, the UC.py script will start a new process on the OS level that will run independently until stopped (see Sec. ?? for details). Setup PVs of a user calc are (with PV name in UC1). As always, the names have no intrinsic significance. Their meaning comes from the relations set up in the database, i.e., the entries in the pertinent columns of the trigger PV.

- UCX.srcPVs (X=1..4): a space-separated list of PV names that enter the expression (below)
- UCX.Expr (X=1..4): the expression to be evaluated by the python `eval()` function. In this expression, \$1, \$2, etc., refer to the PVs listed in UC1.srcPVs, so \$1 is the value that one can read from the first PV in the list in UCX.srcPVs (corresponding X=1..4). There are also special keywords: `_NOW_` refers to the current system time, `_START_` refers to the time the user calc was started, `_PREV_` is the most recent result of the calculation, and `_ACCUM_` is an accumulation buffer for calculating integrals with exponential decay (see below).
- UCX.Dest (X=1..4): the PV to which the result is written by calling `pvw.py`

The user calc is triggered by writing a non-zero value to the trigger PV. That trigger value is used as the update time interval, i.e., how much time passes between evaluations of the expression.

VI. TECHNICAL DETAILS

A. Device Drivers

Drivers translate from the PV structure to device communication. The device-specific communication protocol is built into the respective driver program and device-specific database tables. However, if the device is connected to a bus-type interface (GPIB, modbus, etc.), commands may be wrapped into larger ones that contain bus-routing information (see Sec. II for an example). Drivers correspond to device types, such as Keithley 6487 picoammeter, or DCH motor driver. Information specific to each device is contained in a device-specific database table (DSDT). There is only one DSDT for each device, but, depending on the setup of the database (see Sec. II), multiple devices of one type may be represented in one DSDT. The most common cases would be to have one DSDT for one, and only one, device each, i.e., multiple DSDTs for multiple devices of a given type, or to have just one DSDT containing multiple devices for each type of device. For example, multiple picoammeters of the same type may be present in an experiment. Then, there is one driver program for the picoammeters, and each picoammeter may be represented by one DSDT, or all picoammeters are collected in one DSDT (or a few pA meters are in one DSDT, others in another, etc.). This is illustrated schematically in Fig. II. The setup procedure automatically generates

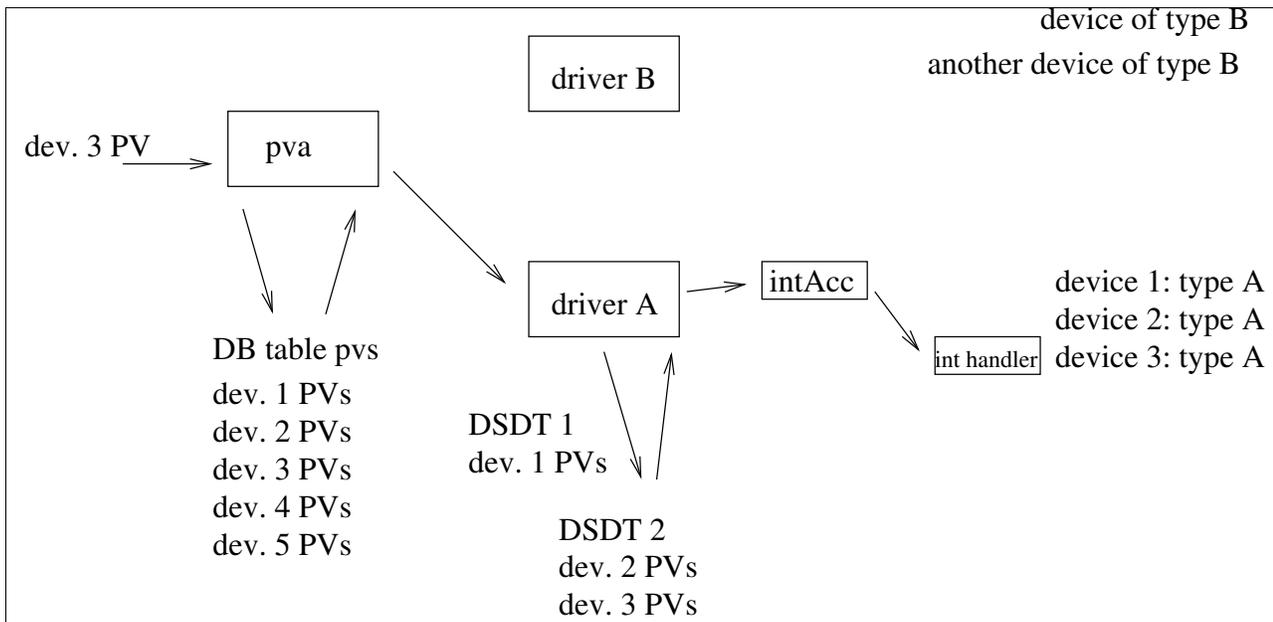


FIG. 4: Schematic representation of device access through drivers: A user accesses device no. 3; `pva` consults DB table `pvs` to find that device no. 3 requires driver A, and that specific information is found in the device-specific database table DSDT no. 2 (which also contains information on device no. 2 that is not required here); `pva` calls driver A and tells it to look for the requested PV for device 3 in DSDT2; with that information from DSDT2, driver A assembles a command and sends it to `intAcc` together with information on which interface to use. `intAcc` then accesses the interface for device 3 through the proper interface handler.

the entries in the PVs database table to direct the driver program to the correct DSDT for each PV access.

The DSDTs contain listings of PVs associated with each respective device along with information what to do when the PV is accessed. The DSDTs are the places where the PVs are specified, as described in Sec. II - each PV is specified in exactly one DSDT description file, and nowhere else in the experiment-setup files.

All driver programs expect either one or five arguments. If the first argument is 'ID', then no further arguments are required, and the driver returns an ID string containing its ID, version, author, etc. information. Otherwise, the first argument specifies which action to take with respect to a device. The next arguments are then: `callfrom`, `dsdt`, `pvname`, `pvtype`. `callfrom` contains a string with call-history information for debugging purposes, `dsdt` is the name of the DSDT, `pvname` is the PV name, and `pvtype` is one of 'R', 'W', 'S'.

VII. ACCESSING PROGRAMS OUTSIDE OF C3PO

In some cases, it may be more convenient to control a device through software supplied with it instead of a C3PO driver. C3PO provides a way for doing so for device-specific software that can be called on the command line and returns results through the command line or data files: For this task, a driver named `ShellProg` is provided along with a template PV definition file `ShellProgram_Defs`. The following explanation will become clear by referencing to the `Defs` file. The template file defines three sets of PVs that serve to define the program name and command-line parameters, and to read back replies from the external program. The names of the PVs are defined in the `Defs` file by the line entries and the line `#define ShP ShellP`, which changes every occurrence of `ShP` to `ShellP` (very useful for global name changes), and by the line `foreach @ in {1,2,3}`, which tells the setup script `setupDB.py` to generate PV names with “@” replaced with “1”, “2”, and “3”. These may easily be changed. The three `ShP` sets can be used to associate any program with PVs, define the command-line parameters, and read back replies. This works through internal PVs, each of which holds one piece of information (such as the program name, or one command-line parameter, each). With this, the PV structure, which allows writing or reading only one item at a time, can be used to build more complex commands. The PVs that read from and write to these internal PVs have associations defined with them that link them to the internal PVs. The same holds for the PV that triggers program execution. In order to use this structure, one needs to issue a series of PV write commands, one for the program path and name (or separately for path and for name), and one for each command-line parameter. When all this is set up, a write operation to the trigger PV (`ShellPx.exec`, where $x=1,2,3$) starts the external program. Reading from the PV `ShellPx.read` will also trigger the external program and wait for its completion, then return a result. That result (also if done through `ShellPx.exec`) is stored in an internal PV and can be read from the PV `ShellPx.result`.

There is also a set of PV in the template file that shows how to pre-define associations of programs with PVs, bypassing most of the internal PVs, and thus saving setup work for programs used often in a particular setup. This works by putting a colon in front of an entry in the PV association field, signifying a direct value instead of a reference to an internal PV.

PV name	type	function	parameter	result	comment
<code>ShellPx.exec</code>	W	trigger	ignored	none	$x = 1..3$
<code>ShellPx.read</code>	R	trigger and read-back	N/A	program result	$x = 1..3$
<code>ShellPx.result</code>	R	return last result	N/A	result of most recent call	$x = 1..3$
<code>ShellPx.prn</code>	W	set program name	name	N/A	$x = 1..3$
<code>ShellPx.prn</code>	R	get program name	N/A	name	$x = 1..3$
<code>ShellPx.prp</code>	W	set program path	path	N/A	$x = 1..3$
<code>ShellPx.prp</code>	R	get program path	N/A	path	$x = 1..3$
<code>ShellPx.prpn</code>	W	set program path/name	path/name	N/A	$x = 1..3$
<code>ShellPx.prpn</code>	R	get program path/name	N/A	path/name	$x = 1..3$
<code>ShellPx.pary</code>	W	set program parameter no. y	path/name	N/A	$x = 1..3, y = 1..10$
<code>ShellPx.pary</code>	R	get program parameter no. y	N/A	path/name	$x = 1..3, y = 1..10$

TABLE IV: PVs for external-program access provided by the template `ShellProgram_Defs` (when reading that file, remember to substitute “`ShellPx`” for `ShP@`, where $x = 1, 2, 3$ is inserted for “@” per the “`foreach`” line). As stated above, PVs with the same name but of different types are different PVs.

VIII. SCRIPTING THROUGH THE PV STRUCTURE

Besides individual-PV access and strictly formalized procedures like scans, it is often necessary to exert more free-style control about processes in an experiment. One way for doing so is to use external programs as described in Sec. II. However, experiment parameters that are accessible through PVs should be controlled from within C3PO. Scripts provide this free-style PV access.

Examples are the before and after scripts for a scan, i.e., scripts that can be optionally defined to execute at the beginning, and after conclusion of a scan. These need to be called from within C³PO, and the only way to do so is to write to a trigger PV for the respective script. There is a dedicated driver script that is activated when one writes to such a trigger PV, and other PVs are also associated with this driver to pass parameters to the script to be triggered.

IX. SCANS

Scans are a little more complex than simple motor moves or detector-read operations. Note that in the following, the word “scan” is used in two ways that should be clear from the context. One is the provision of a scanning capability, i.e, the scan driver, pre-defined scan PVs, etc., and the other is the actual execution of a scan, i.e, the sequential motions and detector readout operations. To distinguish them, the former will be capitalized, i.e., ‘Scan’ means the scanning capability, and ‘scan’ means execution of a scan.

A scan needs to be set-up, i.e., C³PO needs to know the start and end points, total number of scan points, what to move, what to read, etc. Multi-dimensional scans are created by simply having a scan trigger a scan in the same way as if a detector would be triggered. The scans can be linear with evenly-spaced points between the start end end positions, or they can be table scans where freely defined actuator positions and detector acquisition times are given as lines in a table.

From the user perspective, a Scan is a set of PVs that contain setup and status information. In simple cases, when only one set of actuators needs to be moved, and one set of detectors be read at a time, it is sufficient to work through a single set of such Scan PVs. However, there are cases when several scans need to run simultaneously. The most obvious one is that of multi-dimensional scans. Then, the scans of all dimensions are simultaneously active, and each needs its own set of scan PVs.

Setting up a scan by writing to the pertinent PVs is best done by a script that reads the parameters from a file, as demonstrated with the script `setupScan.py` that is part of the standard installation. A file `setupScanD1` with sample scan parameters is also part of the standard installation. After editing the parameter file, one simply calls the `setupScan.py <parameter file>`, e.g., `setupScan.py setupScanD1`, and the database tables will be populated as required. This setup procedure needs to be done each time a scan is to be performed at an experiment. It should not be confused with the installation of the scan PVs, which is done by the script `run_installation.py` once per site installation.

To start a scan, one writes a “1” to a trigger PV. To the end user, this is a PV of type ‘W’, just like any other write-to PV. The scan driver will know from entries in its own database table that this PV is meant to trigger a scan, and which other PVs contain all the scan setup information. Before the scan is started, these PVs need to be populated with the pertinent information, simply by writing to them. Since they are all associated with the scan driver in the pvs database table, the values written to them are passed to the scan driver, which then handles the task of entering the information into its own database table.

The database table `scans` has several columns for scan setup information. By writing to a PV that is of type W in pvs and type S in scans one can fill in these columns. Several of such write operations are required, one for each scan parameter (start, end points, no. of intermediate points, etc.)

To practice scanning, one can set up a dummy scan using FIFO or RW PVs (see Sec. XI) to write “actuator” positions to, and read them back like a detector. Detectors can also be simulated using the software timer (see Sec. ??).

A. Scan PVs

Each Scan needs a set of PVs, as shown below for the `scanD1` that is part of the core scripts of C³PO. These PVs fall into three categories: 1) trigger (type ‘W’, typically one such PV per Scan), 2) setup PVs (type ‘W’) that tell the scan driver how to conduct the scan, and 3) status PVs (type ‘R’) that return information on whether the scan is done, etc. These PVs are brought into relation with each other through database entries created in the site-installation process. Although the end user does not need to create these entries, and does not necessarily need to know about them, it is very helpful to gain some understanding of the way the relations between PVs make a Scan. This is described in Sec. II

PV names are free of constraints, even if some naming system is advised, such as here all PVs beginning with `scanD1`. However, neither the part before, nor after the period in the name is required. The scan scripts do not derive any structural information from a name itself, but only from how cross-references are set up in the scans database table.

- `scanD1.Trig` *triggers a scan once everything is set up*
- `scanD1.ScanPath` *OS path to where the data collected in the scan are written*
- `scanD1.ScanFile` *to where the data collected in the scan are written (path is prepended to this file name)*
- `scanD1.Before` *optional: a trigger PV (type ‘W’, write a ‘1’ to it to trigger. This is done before the scan is executed*

- scanD1.After *optional: a trigger PV (type W', write a 1' to it to trigger. This is done after the scan is executed*
- scanD1.ScanType: 'lin' or just empty for a linear scan, 'table' for a table scan
- scanD1.Actuators: a space-separated list of actuator PV names
- scanD1.ActSettle: a space-separated list of settling times for the actuators given in scanD1.ActSettle
- scanD1.ActFrom: a space-separated list of starting points for the actuators given in scanD1.ActSettle
- scanD1.ActTo: a space-separated list of end points for the actuators given in scanD1.ActSettle
- scanD1.NptOrScFile
- scanD1.DetTrg: a space-separated list of PVs to trigger detectors
- scanD1.DetWait: a space-separated list of PVs that reflect detector status. A detector that is done returns 1, otherwise 0
- scanD1.DetRead

B. Scan Internals

PVs related to scanning are represented as entries in the `pvs` database table, which directs the `pvr.py` and `pvw.py` scripts to call the `scans.py` driver and to (normally) use the `scans` database table. Each of these PVs then has another corresponding entry in the `scans` database table. This latter table has columns by the following names: `pvname`, `type`, `ivalue`, `assoc`, `scanpath`, `scanfile`, `before`, `after`, `scantype`, `actuators`, `actsettle`, `actfrom`, `actto`, `nporscfile`, `dettrg`, `detwait`, `detread`, `comment`. Of these, `pvname` and `type` contain the name and type of each PV, `ivalue` is used only with the setup PVs (see below), and `scanpath`, `..`, `detread` are used only with the trigger PV (see below). PV types are 'T' for trigger, 'S' for setup, 'R' for read-status, and 'I' for 'internal', where 'T' and 'S' correspond to type 'W' in the `pvs` table, and 'R' is also 'R' in the `pvs` table. internal PVs are not represented in the `pvs`, but only in the `scans` table.

The setup PVs are used to send parameters to a scan, such as which actuators to move, which detectors to read, etc., by writing the parameter value to the respective setup PV (of type 'W' in the `pvs` table and of type 'S' in the `scans` table). This has no immediate effect; rather, the value is stored in the 'ivalue' field of the PV entry in the `scans` table. These parameters are retrieved from the database when a scan is triggered by writing a '1' to the scan-trigger PV, which has the following characteristics:

- it is of type 'W' in the `pvs` database table
- has an entry 'Scans' in the 'dname' column of the `pvs` database table
- has a type 'T' in the `scans` database table (or whichever other database table is specified in the `pvs` table)
- has the columns `scanpath` .. `detread` in the `scans` properly populated with the names of PVs containing the scan parameters (setup PVs, type 'S')

When a '1' is written to a scan-trigger PV, `pvw.py` calls the `scans.py` driver (because it is told so by the entry in the 'dname' column of the `pvs` table), and passes the PV name, scans-specific database-table name, and the value to that driver. The scans driver then looks in the scans-specific database table and finds that the PV is there of type 'T'. Following this, the driver looks up the entries in the `scanpath`, `..`, `detread` columns of the entry for the trigger PV. These name the setup PVs where the driver will then find the parameters.

X. SOFTWARE TIMER

The driver `SwTimer.py` provides coarse timing functions through the timing capabilities of the operating system. Depending on the capabilities of the computer, one can expect accuracies between milliseconds and about a second. Precision timing for gating detectors, etc., should be done with dedicated hardware that requires its own separate drivers. The `SwTimer.py` driver can handle multiple timer channels, each of which is represented by a set of PVs in its internal database table `swtimers`. Some of the PVs in such a set contain programmed timer functionality, as

explained below. It is thus easy to add further functionality to a timer by setting up the PVs. The timer driver itself only has two functions: start counting seconds, and doing arithmetic on PVs. At the minimum, each timer channel requires a PV for triggering it (start counting seconds), and at least one PV to read the status. When a timer is triggered by writing to the associated PV, the current system time is captured in a database entry internal to the SwTimer.py driver. Any subsequent status requests will then subtract that start time from the time when the status was requested to obtain the time since trigger.

The timer channel provided with the base installation has one PV (SwTimer1.Count) to read the fractional seconds since start, as well as four PVs that return yes/no information on whether a threshold amount of time has passed since start (see below). There are also four write-only PVs for setting up those thresholds. What happens when a value (anything) is written to the trigger PV (identified as of type ‘W’ in the pvs table and type ‘T’ in the swtimers table) is that the system time (in UNIX systems typically the number of seconds since Jan. 1, 1970, the UNIX epoch) is written to an internal PV. When the PV SwTimer1.Count is read (type ‘R’ in the pvs table, type ‘A’ for ‘arithmetic’ in the swtimers table) is that the swtimer driver subtracts the start time from the present time, and returns that result. This arithmetic operation is not hard-coded into the swtimer driver. Rather, type ‘A’ in the database table allows for the specification of arithmetic operations (actually, anything that can be processed by the python function eval()). The data to go into that calculation are given as PVs, etc. in a space-separated list in the ‘assoc’ field of the database table. In the pre-defined timer of the installation, the full type entry reads A(\$1-\$2). This tells swtimer.py to take the first entry (\$1) in the ‘assoc’ column, which is `__NOW__`, and the second entry (\$2), which is the internal (type ‘I’) PV SwTimer1.Start, then to resolve `__NOW__` to the current time and SwTimer1.Start to the start time (which was stored there by writing to the trigger PV), subtract the current time from the start time, and return that result. There are also four write-to PVs (type ‘S’ in the swtimers table) to set time thresholds, and four arithmetic PVs that return as their result whether the time since timer start has exceeded the threshold (result 0), or not (result 1). In other words, the result has the meaning of “busy”, or “wait for some time to pass”.

Here is a complete listing of the PVs for the timer channel provided in the base installation:

- SwTimer1.Trig (W): The internal timer is reset when a “1” is written to this PV. Actually, the system time (Unix epoch) is stored in the database
- SwTimer1.Count (R): counter status current system time (epoch) minus epoch when SwTimer1.Trig was issued
- SwTimer1.Thr1 (W): event threshold 1
- SwTimer1.Sta1 (R): event status 1; if the timer counts past the value last written to SwTimer1.EvThr1, the read-back from this PV goes from 1 (true) to 0 (false). In other words, this is a sort of “still-busy” function
- SwTimer1.Thr2 (W): analogous to SwTimer1.EvThr1
- SwTimer1.Sta2 (R): analogous to SwTimer1.EvSta1
- SwTimer1.Thr3 (W): analogous to SwTimer1.EvThr1
- SwTimer1.Sta3 (R): analogous to SwTimer1.EvSta1
- SwTimer1.Thr4 (W): analogous to SwTimer1.EvThr1
- SwTimer1.Sta4 (R): analogous to SwTimer1.EvSta1

Other functions based on elapsed time can be set up by populating the swtimers database table, accordingly (or some other database table, which the swtimer.py driver is directed to through the database entry in the pvs table).

To use a software timer, the end user first sets up any timing thresholds (if required), and then sends any value to the designated trigger PV. The elapsed time can be monitored through the designated read-back PV (here SwTimer1.Count), or “busy” PVs (here SwTimer1.Stax).

The pre-defined PVs for the software timer are defined in the file `./installation/coreScripts/CoreTimerDefs`, and these definitions are inserted into the database by the script `./installation/coreScripts/setup_swtimers.py` in the installation process. Other timer channels can be set-up by adding to the entries defined in `CoreTimerDefs` through a user-defined setup.

XI. FIFO AND RW BUFFERS

Read-write (RW) and first-in, first-out (FIFO) buffers for PVs are part of the standard installation. Both types of buffers are handled by the same driver, `Fifo.py`, and they are mostly meant for development and testing purposes. Associated with each “channel” in the RW and FIFO buffers is a pair of PVs, one to write to, and one to read from. Reading from an RW PV any number of times will return the value last written to its associated write-to PV, or an empty string if nothing has ever been written to it. In contrast to this, a FIFO buffer maintains a memory; whenever something is written, it is pushed onto a stack, and reading retrieves (“pops”) the last item written from the stack, exposing the next-to-last written item for retrieval. In the standard installation, the following pairs of RW and FIFO PVs are defined:

- 4 independent channels of RW: `RW.InX` (write-to) and `RW.OutX` (read-from), $X=1..4$
- 2 independent FIFO buffers: `Fifo.PushX` (write-to) and `Fifo.PopX` (read-from), $X=1..2$, using files `fifobfrX` for storage

The memory of an RW buffer is maintained in the database table specific to the `Fifo.py` driver, and the memory for a FIFO buffer is in a file whose name is specified in the table (“`fifobfr`” in the standard installation). One can create additional RW and FIFO channels by specifying them in a file similar to `CoreFifoDefs` and writing them to the database with a script similar to `setup_fifo.py`. These are both found in the directory `./installation/coreScripts/`

XII. HA LAYERS

The Hardware-abstraction layers are structured as layers of program routines with a severe constriction at the interface between user-supplied routines and C³PO-internal routines. At the narrowest layer, HA:0, there are only two routines named `rpv.py` for reading from a PV, and `wpv.py` for writing to a PV. This constriction ensures a unified interface. Below HA:0, system-internal routines branch out in a structure described below, and above HA:1, users are free to define their own structure.

- **HA:> 1** are routines provided by the user in any style and structure. For access to C³PO, they need to interface with HA:1 routines
- **HA:1** are routines provided by the user, meant to interface with C³PO through the HA:0 routines `rpv.py` and `wpv.py`
- **HA:0** are only `rpv.py` and `wpv.py`. These are the interfaces from the user world to C³PO
- HA:-1 is a single routine. It resolves PV alias names, then determines from the PVS database table, which software driver to call, as well as further information for the driver, namely which driver-specific database table to use, and which interface the physical device is connected to. At this level, the interface is referred to by a symbolic name, which is resolved at HA:-3
- HA:-2 are the software drivers. These generate the command strings to be sent to devices and then either pass the command and the symbolic interface name to level HA:-3, either through an operating-system call to program `intAcc.py`, or through a TCP socket that a daemon called `intacc` listens to
- HA:-3 is the program `intAcc.py` or the daemon `intacc`. Both resolve the symbolic interface name by looking it up in the database table “`interfaces`” populated from the file `interfacedefs`, and then write to or read from a tcp socket being monitored by an interface handler
- HA:-4 are daemons or other tcp-socket entities (such as other computers on the network) that handle interfaces. The daemons need to be started up before C³PO is used.

XIII. THE DATABASE

The SQL (structured query language) database is a central component of the control-software suite. It contains several tables with specific information relating to parts of C³PO. End users do not need to access the database directly and do not need to know anything about its structure. However, for debugging purposes it may be helpful to peek into the database.

Databases are organized into tables, each of which has columns pertinent to data for each data item, and data items are shown in rows. The database table that is most immediately relevant for users is called `pvs`. It contains information pertinent to all the PVs available in a given installation. Also common to all installations are the tables `admin`, `alias`, `fifo`, `interfaces`, `scans`, `scripts`, and `uc`. Furthermore, there is one table corresponding to each device driver. The tables are described further, below. A listing of all the tables can be seen at the `psql` command prompt with (at the UNIX command prompt, type `psql <name of database >`, then type `d`).

A. `pvs`

The table called `pvs` lists all the PVs in a given installation. Access to a PV, typically through a call to `rpv.py` or `wpv.py`, internally within a scan, occurs through a call to `pva.py`, which, after possibly resolving an alias (see II and II), looks into the `pvs` table to find: the type (read/write/internal), the driver to call, the name of the database table that the driver should use for further information, and up to four names of programs to call for side effects.

B. `alias`

PV names can be somewhat lengthy, and each action on each device has a unique PV name. This can be cumbersome to use. Therefore, alias names can be defined, which collect several PVs under a common name. Typically, one would associate different actions of the same device with a single alias name, i.e., an alias `xtr` would refer to the different actions (send to some position, query position, query if done moving, etc.) for a translation stage. For each alias name, the table `alias` lists different contexts (program names where the PV access can originate from, as well as whether this is for a read or write access), and which actual PV name this corresponds to.

C.

D.

The `admin` table has two columns: key and value.

The other mandatory table is called `PVs`. It lists all the PVs used in a given implementation. The columns in that table are:

- index no. (integer)
- PV name (string)
- type (string)
- name of driver program (string)
- execution code for driver (string)
- optional: PV and value (separated by space) for side effect no. 1 Whenever the PV in the entry is accessed, the side-effect value is written to the side-effect PV. It is possible to run scripts by writing to a PV, so side effects can be python scripts, which, in turn, can call any other program
- optional: PV and value for side effect no. 2
- optional: PV and value for side effect no. 3
- optional: PV and value for side effect no. 4
- target value when last accessed (string)
- actual value when last accessed (string)
- optional: comment

Any additional information that may be needed to use the PV is contained in the secondary table, the name of which is passed to the driver, and which the driver knows how to use. The target and actual values when last accessed are mainly useful to hold motor positions. When a value is written to a PV, that value is always entered into the target value-field. The actual-value field may be populated from the return value of side-effect no. 1 (if present, see below). When a value is read from a PV, that value is always entered into both the target and actual value fields.

When a PV is accessed, the main HA program looks up the PV name and checks if the requested operation is compatible with the type (if not, it exits with an error message). Then, it looks up the name (to the OS) of the driver program, and passes the name of the PV, optionally the name of a secondary table where further information can be found, and the value (in case of a write operation) to the driver. The structure of the secondary table depends on the type of PV. It is usually defined when a new type of PV comes into use, possibly years after the initial installation. There are four columns for optional side-effect programs. In the unlikely case that more side effects are needed, they must be handled through indirect calls, i.e., one of the four side effects calling more, secondary ones. Side effects may be needed for several reasons. For example, when a motor is moved to an absolute target position, there is no guarantee that it will actually reach it. Then, a side effect could be to call a program that reads back the actual value. This function is reserved for side effect no. 1

XIV. INSTALLATION

Installation of C₃PO for use in a particular experiment may include the following steps:

1. Installation of other software that C₃PO relies on: python with supporting packages, psql with supporting packages, and, for Windows only, GNU core. This needs to be done on every computer that runs C₃PO, or part of it, but only once. See Sec. XIV A for details.
2. Writing device drivers and interface drivers, if those are not yet present. See Sec. ?? for details.
3. Defining PVs. See Sec. XV A for details.
4. Running the python program `run_installation.py` in the C3PO root directory (see Sec. II

A. Software Installation

1. GNU Core

This package provides some core UNIX commands to be available on a windows system, so it needs to be installed only with windows. The path to the coreutils needs to be set manually: Start menu / right-click computer / properties /adv system settings env var /

2. Python

Windows 7: <https://www.python.org/downloads/windows>, Linux: Install perl with your favorite package manager.

Ubuntu: `sudo apt install python-pip`
`sudo apt install python3-pip`

The list below shows all python packages that are needed by C3PO. Install these with `pip install <package name>` and/or `pip3 install <package name>`. The ones that are typically part of a core python installation are lumped together:

- `os`, `sys`, `subprocess`, `time`, `re`, `logging`, `inspect`, `random`, `psutil`
- `psycopg2` for communication with psql (need to first do `sudo apt-get install libpq-dev python-dev`)
- `pyserial` (not `serial`) for the serial-port interface; the package is included as “import serial”, but it `pyserial` needs to be installed instead of `serial`. The latter will complain about XOFF not being defined.
- `ssl`

- `functools` (it may already be included in python, depending on the version installed)
- `psutil`
- `vx111` `pip install -U python-vx111` (`software/python/python-vx111-master`)
- `visa` `pip install -U pyvisa`, needs also `pyvisa-py`
- to talk to Agilent oscilloscopes via USB: install `usbtmc` (install as `python-usbtmc`) and `pyusb`
-
-

Then, to install python or python3 packages, do `pip install <package name>` or `pip3 install <package name>`

To access the database from python, you need to install `psycopg2` Windows: type in a terminal `pip install psycopg2` www.stickpeople.com/projects/python/win-psycopg/ Install serial module `pip install pyserial`

3. Installing the Database

First, the database software must be available to use, i.e., a `psql` server must be running and must be accessible. Then, a the database to be used must be created. In the simplest case, the server runs on the localhost (the computer that is controlling the experiment) on port 5432.

4. Linux

- install `psql`: `sudo apt install postgresql postgresql-contrib`
- become `postgres` superuser `sudo -u postgres psql postgres`
- on the `postgres` prompt `#: \password postgres`
and enter `psql admin pwd` <https://serverfault.com/questions/110154/whats-the-default-superuser-username-password-for-postgres-after-a-new-install>: Do not set the password for the UNIX account “postgres”
- on that same `postgres` prompt: `create database <dbname>;` Enclose name in double quotes to be sure the capitalization is correct and finish line with a semicolon. If the database name contains capital letter, enclose it in double quotes
- on that same `postgres` prompt: `create user <username> password '<mypassword>';` and finish line with semicolon
- on that same `postgres` prompt: `grant all privileges on database <dbname> to <user>;` and finish line with semicolon
- leave `postgres` with `\q`
-
-

Get `pgadmin3`: (`apt-get install pgadmin3` or through synaptic)

5. Windows 7

6. Installing psql on the W7 computer

- Go to <https://www.postgresql.org/download/windows>
- click on the first link “Download the installer”
- select version 9.4.11 (other versions will probably also work)
- select operating system Windows x86-64
- click the big button “DOWNLOAD NOW”
- once the .exe file is downloaded, open it to go through the installation process. It will ask for a port - use default 5432 You will also need to create a password for the database administrator
- At some point, it will ask for add-ons. Select EDB language pack (includes perl), and optionally others such as apache/php and apache/httpd

7. Creating the database on the W7 computer

- open the pgAdmin program
- double-click in the tree on the left postgresql 9.4(localhost:5432) and enter admin password
- the tree will now list the existing databases, etc.
- click on Login Roles to create a database user:
 - “properties tab; Role name: enter a name, say “xxx”
 - “Definition” tab; password, uncheck expiration data
 - “Privileges” tab: check login, inherits, can create DB
 - “SQL” tab: should list SQL command to be executed
- click on databases in the tree, then Edit tab - New Object - Database

B. Operating-System Setup

1. USB

USB devices do not have firm associations between the physical device and the names by which they known to the OS. For example, USB serial ports on a Linux system tend to appear as `/dev/ttyUSBx`, where `x` is a sequential number in the order the ports were plugged in. If they are already plugged in when the computer is started, they may come up in any random order. This uncertainty can be resolved by assigning them symbolic names that are tied to unique characteristics, such as serial numbers. To do so, create a file `80-SerialPort.rules` (or some name like it ending in `.rules`) in the directory `/etc/udev/rules.d/` with contents that look like this:

```
#FTDI
ATTRS{idVendor}==''0403'',ATTRS{serial}==''A1025EFI'',ATTRS{idProduct}==''6001'',MODE==''0666'',SYMLINK+
==''DCH-usb'',GROUP==''dialout''
ATTRS{idVendor}==''0403'',ATTRS{serial}==''FT3W5E11'',ATTRS{idProduct}==''6001'',MODE==''0666'',SYMLINK+
==''K6485-usb'',GROUP==''dialout''
#prolific
ATTRS{idVendor}==''067b'',ATTRS{idProduct}==''2303'',MODE==''0666'',SYMLINK+=''K2400-usb'',GROUP==''
dialout''
```

The identifying specifics of each USB device (vendor ID, serial no., etc.) can be found by running the command `udevadm info --name=/dev/ttyUSB0 --attribute-walk > capturex` and then looking for those items in the file `capturex`. Once that file is written, run `sudo udevadm control --reload-rules`

For more on writing udev rules, see for example http://www.reactivated.net/writing_udev_rules.html
 Then, make sure a group `dialout` exists on the computer and make the current user a member of that group.

To view all groups on a system: `getent group`

To create a group: `sudo groupadd <group name>`

To see if a user is a member of a group: `groups <username>`

To make a user be a member of a group: `sudo useradd -G <group name> <user name>`

For groups in UNIX, see for example <https://www.howtogeek.com/50787/add-a-user-to-a-group-or-second-group-on-linux/>

C. `run_installation.py`

a. *The master installation script `run_installation.py` performs the following tasks:*

read the file `./setup`

call: `./installation/run_installation.py`

call: `./<customFilePath>/run_installation` (where `customFilePath` is given by the file “setup”)

copy: `cp ./<customFilePath>/C3PO/* ./C3PO/`

copy: `cp ./<customFilePath>/user/* ./user/`

call: `./C3PO/generatePVlist.py`

call: `./C3PO/SystemSetupAll.py`

copy: `./C3PO/userinfo ./user/`

b. *The subsidiary installation script `./installation/run_installation.py` will:*

read the file `././setup`

attempt to create the database (“fails” if already exists)

`mkdir <userdir>` (from setup file: `./user`)

`mkdir <systemdir>` (dto `./C3PO`)

`mkdir <logdir>` (dto)

`mkdir <datadir>` (dto)

copy: `cp ./coreScripts/* <systemdir>` (typ. `C3PO`)

create file `userinfo`

create file `admindefs`

insert the shebang (from setup file) in all `.py` files in `systemdir` and `userdir`

c. *The subsidiary installation script `./CustomFilesForXXX/run_installation.py` will:*

read the file `././setup`

append to file `admindefs`

d. *The subsidiary installation script `./C3PO/generatePVlist.py` will:*

Create a file `Pvs_Defs.user` and populate it with PV information from all files with names conforming to the regex pattern `_Defs.*$`, i.e., ending in `_Defs(something)` (except, of course, itself)

e. *The subsidiary installation script `./C3PO/SystemSetupAll.py` will execute all files with names conforming to the regex pattern `_Defs.*$` These files must have a shebang pointing to `setupDB.py`, the program that generates all database entries.*

D. Database definition files `_Defs*`

The database is populated from information contained in files ending in the regex pattern `_Defs*$`. These files must have the following structure (see example below):

- the first line must contain the shebang `#!/./setupDB.py` to be properly called by the installation script `./C3PO/SystemSetupAll.py` (see above).
- the next line must contain a hash-mark followed by the names of the fields in the respective database table, the name of which is derived by lopping the ending `_Defs*$` the current file.
- the third line must contain a comma-separated list of numbers for the field sizes in the database table. The number of comma-separated items must be the same as the number of field names in the line above

- if the next line contains the phrase `drop old table`, then the respective table is dropped (removed), then re-created empty, then filled with the entries of the current file. Otherwise, the old table is overwritten, possibly leaving old entries in there.

To give an example, here are some lines from file `interfaces_Defs`:

The first 5 lines are:

```
#!/./setupDB.py
# name, dnPort, upPort, driver, start, call, daemon, comment
# 32, 64, 64, 32, 128, 256, 32, 256
# drop old table
# comment: ##
```

E.

1. Database Manual Access

The database can be accessed manually with the command `psql -d xcap`. It is helpful to do so now and again just to check on how the control software has modified the database. Databases that exist for the current user are listed with the command (on the psql prompt `xcap=>`) `\d`. The structure of table `pvs` is displayed with `\d pvs`. All entries in table `pvs` can be shown with `SELECT * from pvs`.

F. Experiment Setup

To set up an experiment, one needs to create the database structure. This could be done manually on the postgresql prompt, but it would be a very tedious process. Instead, one can run a setup python script. However, to alter an existing setup, for example, to set-up a new device, may be easier to do on the psql command prompt.

Entries into the table are made with the `INSERT INTO` command at the database command prompt. For example, `INSERT INTO pvs (pvname,type,drname,sectable) VALUES ('htr.mva', 'W', 'DCH', 'DCH');`

XV. EXAMPLE

Suppose the experiment requires to drive three motors using the DCH motor controller from Dragonfly Devices, as well as reading images from a CCD camera, and controlling a temperature. The motor driver is connected to serial port `/dev/ttyUSB0`, the camera requires a call to a program that handles exposure, readout, etc., and the temperature controller is connected by RS485 bus to an ethernet-serial bridge. Assume the motors are called `htr`, `vtr`, and `foc`, and we use a somewhat structured naming convention for the pVs (this is not necessary, but it is helpful). For each motor, we want to have PVs to send it to an absolute position, to a relative position, read back the current position, and read back the limit-switch status (many more PVs may be desirable).

Then, for each motor, one would require at least PVs to send it to an absolute position

A. PV definition

Each PV is represented in two database tables, the overall listing of PVs in a table called `pvs`, and a device-specific table of some suitable name. So, for example, if there are two pA meters of identical make, then there might be tables called `pameter_a` and `pameter_b`, and all PVs relating to a particular pA meter are listed in the respective table. To populate these tables, one has to write a file that lists in each comma-separated row a PV name and other information to go with it. Just what information that is depends on the specifics of the device. The first line of this file needs to have a shebang `init` that indicates a program that reads the PV info in the file and populates the database with it. The name of that file has to be `<name-of-database-table>_Defs`.

XVI. INSTALLATION

To install C³PO for a particular experimental setup, one needs to do the following:

- Create or edit files in the `/CustomFilesForXXX/C3PO/` and `/CustomFilesForXXX/user/` directories, where XXX is the name of the experimental setup. These are meant to hold any experiment-specific scripts, setup files, etc., that are not part of the standard installation. The files in `/CustomFilesForXXX/user/` are meant to be accessible to the experiment operator, and the ones in `/CustomFilesForXXX/C3PO/` are not.
- edit the file `/setup` to define the name of the database, the path to the experiment-specific directory `/CustomFilesForXXX/`
- run the script `/run_installation.pl` This script reads the file `/setup`, and then calls the script `/installation/run_installation.pl` (same name, but one directory down), which sets up the directory structure, copies the core scripts, and modifies them as needed. Then, `/run_installation.pl` copies the experiment-specific files from `/CustomFilesForXXX/C3PO/` and `/CustomFilesForXXX/user/` to their respective locations, and finally calls scripts `/C3PO/SystemSetup_all` and `/C3PO/UserSetup_all` that create database entries, etc. `/C3PO/SystemSetup_all` is part of the core installation, and `/C3PO/UserSetup_all` must be supplied in the `/CustomFilesForXXX/C3PO/` directory by the person who sets up the experiment.

A. The file `~/setup`

The file `/setup` contains information that guides the installation process. The following entries need to exist:

key	example value	explanation
username:	xxx	UNIX user name
database:	xcap	name of the postgresql database
dbpasswd:	Xcap!	password for access to the postgresql database
customFilePath:	CustomFilesForXcap	the path referred to above as CustomFilesForXXX
user directory:	<code>../user/</code>	where the operator-accessible scripts are to go
system directory:	<code>../C3PO/</code>	where core scripts are to go
data directory:	<code>../DataFiles/</code>	where data files go
error directory:	<code>../logs/</code>	where errors are logged
logging directory:	<code>../logs/</code>	where logs are to be written
precision:	1000	
tracebacklimit:	2	
shebang:	<code>!/usr/bin/python</code>	the shebang line at the start of all python scripts
hostname:	localhost	
pemfile:	<code>../C3PO/test.pem</code>	
crtfile:	<code>../C3PO/test.crt</code>	
parameter separator:	<code>\xaa</code>	
mode:	simulation, debug	

If the keyword “normal” appears in the mode line, C³PO will actually move actuators and read detectors
 Only if the keyword “” appears in the mode line will C³PO actually

B. Installation Example: X-ray Optics Characterization

The installation comprises of three motor drivers with four channels, each, and a cooled CCD camera. The path for CCD image data is `/ImageFiles/`, and it is specified through the admin database table

C. Installation Example: Adding CAEN N1470 device

The following steps are required:

- create the file **CustomFilesForXXX/C3PO/CAEN1470.py**. This is the driver file that handles the communication with the device. It can be copied from an existing driver file, such as `Keithley24xx.py`, and then modified in the section that says “device specific”
- create the file **CustomFilesForXXX/C3PO/setup_CAEN1470.py**. This python script gets called in the installation process and populates the database with stuff specific to the device, which it reads from a `_Defs` file, see below. It can be copied from another setup file, and modified in the sections that say “specific”
- create the file **CustomFilesForXXX/C3PO/CAEN1470_Defs**. This file is read by the setup script (item above), and contains the information that is written to a dedicated table in the database, which is created by the `setup_` script. The format of this database table, i.e., the number of columns and their type, meaning, etc., is not determined by the C3PO standard, and can be chosen according to the needs of the particular device. Each row in the table contains all the information that is required for one PV.
- if the communication interface is USB, then one will want to define an alias name for the interface, and enter that into the `interfacedefs` file. See above for the purpose of this
- modify the file **CustomFilesForXXX/C3PO/UserSetup_all.pl**, by adding a line `system($thisdir.”setup_CAEN1470.py”);` this tells the installation scripts to include in the installation the PVs, etc. related to the device
- modify the file **CustomFilesForXXX/C3PO/UserPvsDefs** to enter all the PVs for the driver

1. The Device-Specific Database Table

In this example of the CAEN 1470 device, the following columns are required in the database table:

- PV name; for each of the 4 channels in the CAEN, there are multiple PVs, such as for setting a voltage, turning the channel on/off, etc, as well as for reading from it
- PV type (R/W/S)
- action; this is a string that tells the driver program what to do with this PV
- interface name (as known to the OS, or an alias to be resolved through the `interfacedefs` file)
- board: which CAEN device in a daisy chain
- channel number (0..3)
- communication parameters; these don’t really need to be given here because the CAEN talks only with specific parameters (9600, 8N1, xon/xoff), so they could be hard-coded into the driver. For consistency with other devices, they are provided through the database, anyway
- comment

XVII. ADDING A NEW DEVICE

- write a device driver (if not already available)
- define PV names to represent the functionality
- define name for database table with PVs for this device
- create entries with these PVs in file `UserPVs_Defs`, where each line has entries for: PVname, type, driver, table, side1, side2, side3, side4, comment
- create file `>device-name>_Defs` and enter lines for the PVs in a format understood by the device driver
- write a program (to be entered in the shebang of the `>device-name>_Defs` file, that reads it and populates the database from it

A. Writing a Device Driver

A device-driver program must accept the following command-line parameters:

- `argv[1]`: action (ID—VER—DATE—AUT—DBT—EXEC). All but the last

XVIII. SPECIFIC INTERFACES

Interface and devices connected to them are defined in the database-setup file `interfaces_Defs`, which is copied into the `interfaces` database table when the installation script `run_setup.py` is run. This file contains entries for physical interfaces on the host computer, such as serial ports, USB ports, LAN, etc. - basically any device that would have an entry in the `/dev` directory of a linux system. Each of these entries has a unique name, which can be used directly to access the corresponding device (by being defined as the interface in the setup file for a device-specific database table DSDT). However, usually, there are additional named entries, which refer to actual interfaces. This indirection facilitates the assignment of physical interfaces to devices, which is all done in the `interfaces_Defs` file instead of scattered device-definition files. More importantly, it allows for additional parameters, protocol wrappers, etc. to be passed to the interface drivers.

It is possible, and quite usual, for several device entries to refer to the same interface. The most common occurrence of this multiplicity is when an interface is not connected directly to a device, but rather to another interface, such as a serial-to-GPIB converter. Then, multiple devices would be on the same GPIB bus (each with its own GPIB address), but as far as the host computer is concerned, they are all on the same interface (the serial port). In order to handle multiple devices on one host-computer interface, each device entry contains wrapper information that is meant for the secondary interface on the host-computer interface.

The database-setup file contains entries in the following columns: `name`, `dnPort`, `upPort`, `driver`, `start`, `call`, `daemon`, `comment` with the following meanings:

- `name` is the name of a device or an interface (depending on the type of entry),
- `dnPort` is the communications downlink. For device entries, this is the name of the corresponding interface, which is given in another entry in `interfaces_Defs`. For interface entries, this is the physical interface in the computer, such as a serial port.
- `upPort` is the communications uplink, i.e., where in C³PO communication originates from. For device entries, this field may be empty because device communication always originates from the C³PO-internal program `intAcc.py`. There may be an entry like `_intAcc_` in this field, but that is ignored and serves only to remind the person editing the file of how the communication flow goes. For interface entries, `upPort` is the port that C³PO uses to access the interface, i.e., the TCP port for SSL daemons.
- `driver` is ignored for device entries and is the name of the daemon program in the case of interface entries
- `start` is ignored for device entries and contains startup parameters for the daemon process, such as communications parameters for a serial port
- `call` is ignored for interface entries and contains wrapper information for device entries
-

For example, a serial-to-GPIB converter needs to be given specific commands to write to and read from the GPIB bus, as well as the specific GPIB address of the device. The wrapper information contains four items, which may be names of command-line programs to modify strings or perl-style regular expressions, which may contain keywords recognized by the interface driver. They are separated by a character given as the first non-whitespace character of the wrapper string. If the next two characters following that separator are non-whitespace, they are interpreted as the opening and closing limiters for a command-line function call to modify a string. Otherwise, these are by default the opening and closing parenthesis. Typically, the item separator would be the semicolon, but there may be a case where the semicolon is needed as part of a command for the secondary-interface driver. In such a case, one would use another character as the separator. What C³PO does is to split the string with the wrapper information on the character that is the first non-whitespace of the wrapper string. This yields four substrings (some or all of which may be empty), each of which can contain zero, one, or more function names (in the above-mentioned enclosing limiters) or regular expressions. These multiple sub-items may be defined in an

entry for an interface driver, or may be assembled from multiple indirections of protocol wrappers leading to an interface driver. In the latter case, the sub-items are separated by a non-printable character defined in the C³PO setup file. Here are two examples:

a. Wrapper for an Agilent E2050 LAN-GPIB converter: This wrapper would be contained in the entry for a device that is referenced to the SSLlanGPIB interface driver written for Agilent E2050 LAN-GPIB converters: ; /__GPIB__/9/ ; ; ;

Here, the leading semicolon identifies the semicolon as the field separator for the wrapper information. The next field “all” contains a regular expression saying to replace every occurrence of __GPIB__ (internally in the SSLlanGPIB driver) with a “9”. In other words, this tells the SSLlanGPIB driver to insert the GPIB address 9 where appropriate, where __GPIB__ is a keyword recognized by the SSLlanGPIB driver program. The other items in the wrapper information are empty because the SSLlanGPIB driver is rather specialized, so it knows what else to do.

b. Wrapper for a National Instruments serial-GPIB converter: The serial driver is much more general-purpose than the SSLlanGPIB driver. It is therefore necessary to tell it more about how to do, for example, GPIB communication through a National-Instruments serial-GPIB converter. The wrapper for this configuration would be:

```
; /__GPIB__/9/ ; /(.*)/wrt#__NWRITE____GPIB__\r$1\r/ ; /(.*)/rd#__NREAD____GPIB__\r\n/ ; /x00+.*// $
```

Following the semicolon to set the item separator, there is, again, the regular expression for inserting the GPIB address 9 for the driver-internal keyword __GPIB__. The next item is a regular expression to prepend the string wrt__NWRITE____GPIB__\r, as well as append \r to any sting to be written on the GPIB bus (you need to be fluent in perl-style regex to understand what is going on here). The prepended and appended parts are meant for the serial-GPIB converter, and are not written out to the GPIB bus. The driver program will know how many bytes to write and insert that length for __NWRITE__, and it will also do the regex replacement of “9” for __GPIB__.

If the secondary interface is changed out, then the references from devices to interfaces need to be modified, as well as wrapper information.

A. RS232

Communication through RS232 or RS422/RS485 serial ports is handled by SSLs232 daemons. For each open port, one instance of the daemon is running, but each with its own unique TCP/IP port. Thus, each serial port presents itself to the rest of C³PO as a TCP port. An RS232/RS422/RS485 port may be connected directly to an instrument, or there may be another interface device, such as a serial-to-GPIB interface. Some instruments may also have a USB connection that goes to a built-in USB-serial converter. To the operating system, such a USB connection appears as a serial port, and is thus handled by the SSLs232 interface daemon. As stated above, each device is represented by an entry for itself, and one for the interface.

The entry for the interface has the following columns:

int.name, OS device name, TCP/IP port, SSLs232.py, comm. pars, , , , comment

For example:

```
SSLs232.3, /dev/ttyUSB2 , localhost 12347, SSLs232.py, 9600 8 N 1 xoff 1 0.1, , , # comment
```

Here, SSLs232.3 is the C³PO-internal name of a particular serial port, known to the operating system as /dev/ttyUSB2. The daemon is to run on the same computer where C³PO is running (localhost), and is to use TP port 12347. The program for the daemon is SSLs232.py, which is started automatically upon first access to the serial port, and which will then run continuously in the background, listening for communication requests. It will stop either when told so explicitly by writing a “1” to the PV daemon.stop, or if stopped by the operating system (or if it crashes). Recovery from a crash is described below. The next entry defines the communication parameters, here 9600 baud, 8 bits, no parity, 1 stop bit, using the xon/xoff protocol and timeout parameters 1 and 0.1. The first of these tells the daemon how long to wait for the first byte of a reply after a service request (here 1 s). The other one tells it how long to wait for each subsequent byte. It is assumed that a device may take some time to process a service request (here up to 1s), but then, once it begins transmitting an answer, the bytes follow one after the other. A typically much shorter break (here 0.1s) indicates that the device is done transmitting, so there is no need to wait for the longer timeout used at the beginning of the reply string.

Another entry in the database-setup file makes the link to the device connected to the interface. This entry has the following columns:

device name, interface name, (ignored), (ignored), (ignored), wrapper info, (ignored), comment.

The (ignored) columns play a role

XIX. NOTES ON SOME DEVICES

A. Metrabyte/Keithley/Omega/DGH transmitter modules

These devices use serial communication through either RS232 or RS485 (depending on the model number). For RS232/RS485 wiring, refer to document Omega M0662.pdf. The RS485 devices will also communicate with RS232 (see manual), but this is recommended only for setup, not for normal operation. As long as the default pin is connected to gnd, the device is in a standard state of communication settings: 300 baud, 8N1, will accept any device address. The settings for normal operation can then be changed with the SU command with the following steps:

1. start serial-communications program, such as minicom and set communication parameters to 300-8-N-1 and no handshake
2. ground the default pin
3. type #1RS to test communication and read the current setup The response is a string like *1RS310200408A, where 31020040 is the current setup and 8A is a checksum
4. compose the setup string consisting of four hex-coded bytes: byte 1 is the unit address; for example a unit address of one would be hex for ASCII '1', i.e., 31; byte 2 configures linefeeds, parity, baudrate. Refer to table 5.2 in the manual file M0662.pdf. For lf, no parity, normal addressing and 9600 baud, byte 2 is '82', for 19200 baud it is '81', for 38400 baud it is '80', for 57600 baud '89', and for 115200 baud '88'; byte 3 sets alarms - here we will set all to 0; byte 4 sets the displayed digits and filter time constants - refer to table 5.4. To display all digits and apply no filtering, the byte is C0. The complete setup string is then for this example (9600 baud): 318200C0
5. enable writing to the device by typing \$1WE (write-enable, this is active only for one write op)
6. type the setup string \$1SU318200C0 (....00C2 for 1712 dig.IO)
7. type \$1RS to read-back the setup information\$
8. power-cycle

For straight-out serial communication, the entries into the `interfaces` database table (file `interfaces_Defs`) are rather straightforward, for example: `M3171, SSLs232_10, , , , ;() / / ; ; , 0, # SSLs232_10, /dev/ttyUSB0 , localhost 12360, SSLs232.py, 9600 8 N 1 xoff 1 0.1 fa wrd0.2, , , # comment`

When connected to an EIS-W ethernet-serial bridge, an RS485 unit will now respond. For example, if the unit address was set to 2, comm. pars to 9600-8-N-1, etc. (see above), and if a telnet connection has been made to the bridge at port 2000, then typing \$2RS will respond *328200C0

B. Agilent E2050A

The Agilent E2050A/B is an internet-GPIB bridge, which is rather straightforward to use with C3PO. The device is represented in the `interfaces_Defs` file with a line, such as:

```
SSLlanGPIB_0, 192.168.1.240gpib0, localhost12340, SSLgpibAg2050.py, , , , #AgilentE2050
```

where `SSLlanGPIB_0` is the name by which the interface device is known to C3PO (one unique name for each device in an installation), 192.168.1.241 is the IP address of the device, `gpib0` is its symbolic name required by the VX11 communication libraries, and `localhost 12340` are the TCP host and port under which the `SSLgpibAg2050.py` driver runs. The IP address and the symbolic instrument name are assigned to the device as follows (manual HP_E2050.LAN.pdf, p. 50):

- power on
- press the recessed “Config Preset” button on the rear. The “Conn” LED on the front side will now flash slowly, and the IP address will be at its default 192.10.0.192
- make sure the computer has access to the 192.0.0.255/24 network

- type telnet 192.0.0.192 (return) to connect to the device
- a listing of possible of commands will be displayed
- To set the IP address, type ip: xxx.xxx.xxx.xxx (address)
- To set the symbolic instrument name, type hpib-name: <name>
- type exit to save and exit telnet
- power-cycle to activate new IP address

C. Agilent E5810A

The Agilent E5810A/B is an internet-GPIB bridge. It is a lot larger than the E2050 and provides some extra functionality over the latter. In particular, it has a built-in web server where one can configure parameters and test communication with GPIB attached instruments. It is also a lot more expensive. In C3PO it is configured and used in the same way as the E2050, except for this detail: In order to set a default IP address, press the recessed “preset” button on the front side for more than 10 seconds (pressing less than 10 seconds will only reset the password for the web server). The device will then re-boot and, after some time, will display its IP address on the LCD display. From this point on, configuration proceeds as described for the E2050.

D. National Instruments GPIB-ENET

The NI GPIB-ENET is an ethernet-GPIB bridge. https://sigrok.org/wiki/National_Instruments_GPIB-ENET It uses proprietary NI software and is currently not supported under C3PO. Apparently, the device uses rarp to get an IP address. On ubuntu:

- install rarpd on the linux box
- create or edit a file /etc/ethers with a line containing the MAC address and intended IP address, such as
00:80:2F:FF:41:C9 192.168.1.61
- `sudo /usr/sbin/rarpd -a`
- set all DIP switches to “off” (up) and connect ethernet to GPIB-ENET (see below for switches)
- turn on the GPIB-ENET device
- telnet <IP address > 5000
- once the IP address has been obtained through rarp, switches 5 and 6 can be turned on to continue using that IP address without rarp

The DIP switches have the following meaning (<http://www.ni.com/pdf/manuals/321243d.pdf>, p. A-1): 1-4 reserved (need to be off), 5: off/on: auto/manual IP assignment, 6: off/on: obtain IP addr./use stored IP addr., 7: off/on: normal/modify firmware mode, 8: off/on: normal/factory test

E. Keithley 487

The Keithley 487 is handled by the driver Keithley48x.py . The following list contains the commands that the driver accepts (capital letters), commands that the Keithley instrument will receive (compare Keithley manual Table 4-1), typical implementations as PVs (where k487a is an example of an instrument name), and parameter descriptions (P means parameter):

- LIST, k487a.list . This command is not sent to the instrument, but rather lists all the PVs defined for this instrument.
- RESET, k487a.reset P, if P==“1” or P==“on” (upper case also ok) send command “O0L2X” to the instrument, meaning “put voltage source into standby” (O0) and “revert to saved defaults” (L2), otherwise do nothing

- INIT, k487a.init P, if P==0 send command “L0X” (return to factory default and save), if P==1 send command “L1X” (save present state as default), if P == 2 send command “L2X” (return to saved default), otherwise do nothing (instrument commands L3..L6 are not implemented)
- HKEY, k487a.init P, for P an integer between 1 and 17 (incl.), send command “HPX” to “hit” a key on the front panel (see manual), otherwise do nothing
- DISP, k487a.disp P, display message “P” on front The message is cleared and the display is returned to normal with the BRGT command (next below)
- BRGT, k487a.brgt P, if P == 0, display at full brightness, if P==1, dim, if P==2, off (except power button), if P==”C” cancel displayed message
- TEST, k487a.test P, if P==0,1 perform test 0 or 1 (see manual), otherwise do nothing
- BUFS, k487a.test P, if P==1, .. 512, set data buffer size to P, otherwise do nothing
- DFOR, k487a.dfor P,
- ZCHK, k487a.zchk P, perform zero check for P==”2” or P==”zchk.do”, enable zero correction for P==”2” or P==”zchk_ena”, disable zero check for any other parameter values
- RANG, k487a.rang P, set range in Amps, will choose among available ranges of 2e-9, 2e-8, .. 2e-4 A
- FILT, k487a.filt P
- AVER, k487a.avg P, if P == “1”, “line” (or upper case), send command “S1X” for line-cycle integration, otherwise send “S0X” for fast integration
- INTV, k487a.intv P, set the time interval between triggers
- TRIG, k487a.trig P, if (regex perl style) P = m/[MS][TGXEO]/, set trigger condition to Multiple/Single and to trigger on Talk, Get, X, ext. trigger, or operate voltage source (487)
-
-
-
-
-

F. Omega CNI-Series temperature/process controllers

Before a controller can be used, it must be set up. The set-up parameters are specified with the commands listed below. These do not write directly to the controller, but only to the database. The “init” command, i.e., writing to the PV TconX.init (see below) reads the setup parameters from the database and sends them to the controller. This is illustrated in the example in Sec. II.

1. Input

The controllers accept thermocouples, platinum resistors, or process voltage/current signals on their input.

PV name	type	value	description
TconX.input	W	RTDxxxx , xxxx= 100, 500, 1000	set to RTD input 100/500/1000 Ohm
TconX.input	W	392.2 392.3 392.4 385.2 385.3 385.4	type of resistor (default 392.2)
TconX.input	W	100mV 1V 10V 20mA	set to process input
TconX.input	W	J K T E N DIN-J R S B C	set thermocouple
TconX.input.rb	R		read input register in database
TconX.ainput	R		read input register in controller

An RTD can be specified in two steps, i.e., with the command `wpv.py RTD1000`, `wpv.py RTD500`, `wpv.py RTD100`, or just `wpv.py RTD` for `RTD100`, and then the type of resistance curve, i.e., `wpv.py 392.2`, etc., or it can all be done in one step, for example `wpv.py RTD100-392.2`. What matters is only that it is a single string containing the pertinent substrings. The `type-of-resistor PV` only modifies the bits for the type of resistor or thermocouple, but does not change to `RTD`. One should not specify this parameter after setting a `TC` because it will affect the type of `TC`.

2. *Output 1*

Output 1 is typically used for controlling a feedback loop, i.e., for example, to control an SSR. Alarm 1 must be turned off for output 1 to be useable.

3. *Output 2*

Output 2 is typically used for alarm, emergency shutoff, etc. Alarm 2 must be turned off for output 2 to be useable.

4. *Alarm 1*

5. *Alarm 2*

G. **Omega iServer Microserver**

1. *EIS-W*

The EIS-W has a default IP address of 128.100.101.254/24, which is activated by sliding the 2nd microswitch on the bottom to “on”.

Then `http://128.100.101.254`, select “iServer” from the drop-down menu and then click “Update”. Click “Login” and use the default password 12345678 or the admin-login password 00000000

First click “Access control and set the IP address, then power down the EIS, deselect default on the dip switch, and power on. Then click “Configuration” and select the serial-communication parameters. For server type, select “slave” and 0 sockets. Remote access “enable”, TCP port of 2000

Alternatively, `telnet 128.100.101.254 2002` or

[1] This is an example of the point made in the introduction that much of C3PO is not present in program code but rather in the database structure and contents.

[2] there is nothing in there that would prevent a scan from triggering itself, but it would not be wise to do so.