ISBN: 978-3-95450-237-0 ISSN: 2673-5512 doi:10.18429/JACoW-PCaPAC2022-FR012

AUTOPARAM, A GENERIC ASYN PORT DRIVER WITH DYNAMIC PARAMETERS

Jure Varlec*, Cosylab d.d., Ljubljana, Slovenia

Abstract

Implementing EPICS device support for a specific device can be tricky; implementing generic device support that can integrate different kinds of devices sharing a common interface is trickier still. Yet such a driver can save a lot of time down the road. A well-known example is the Modbus EPICS module: the same support module can be used to integrate any device that speaks the Modbus protocol. It is up to the EPICS database to map device registers to EPICS records. Because no changes to the driver code are needed to integrate a device, a lot of effort is saved. At Cosylab, we often encounter device controllers that speak bespoke protocols. To facilitate development of generic drivers, we wrote the Autoparam EPICS module. It is a base class derived from `asynPortDriver` that handles low-level details that are common to all generic drivers: it creates handles for device data based on information provided in EPICS records and provides facilities for handling hardware interrupts. Moreover, it strives to provide a more ergonomic API for handling device functions than vanilla `asynPortDriver`.

INTRODUCTION

When it comes to putting together a control system, one of the advantages of EPICS is that, given existing device support, integrating devices requires little or no programming: data that devices provide or consume is mapped to process variables *declaratively* through *records* in an EPICS database [1]. Because EPICS is not merely software, but a vibrant community, lots of existing device support modules are readily available [2]. Chances are, then, that integrating a widely-used type of device into your control system requires very little effort.

Of course, not all devices are widely used. Some are brand new, some are niche, some may even be developed specifically for a particular machine. In such a case, developing a new device support layer cannot be avoided. This is an arduous task: device support represents a mapping from device functionality to records. This means that a device support layer is required for *each record type* of interest [3]. Much of this work is repetitive and results in pretty much the same record-specific code across many different device types. For this reason, the device support layer of EPICS lends itself well to encapsulating the repetitive common code in a reusable module.

`asynDriver` [4] is such a module. It has become the go-to module for integrating new devices, and even whole data processing pipelines [5]. Instead of implementing EPICS per-record device support directly, one instead implements one or more `asyn` *interfaces* to wrap the low-level device driver or communication protocol. These interfaces are then used by `asyn`'s *generic* per-record device support layer which covers pretty much all record types that are meant to get data into or out of hardware devices. By using `asyn`, one is thus saved from a fair amount of repetitive and error-prone work: write one driver, support all records automatically. The most straightforward way to create a device support module based on `asyn` is to create a C++ class derived from the `asynPortDriver` base class.

One can go a step further and recognize that some devices are themselves "generic" in the sense that they are merely front-ends to other devices. One example are programmable logic controllers (PLC) which often serve as interfaces to sensors and actuators. It is thus natural to desire an EPICS module that would communicate with a particular type of PLC in a generic enough manner to facilitate integrating it into the control system regardless of which peripherals are connected to it. There are, in fact, several such EPICS modules available, the Modbus module [6] being a very nice example. It allows integration of any PLC (or other device) that speaks the Modbus protocol. With a device support as generic as this, the "no programming" ideal mentioned in the beginning is truly possible: the EPICS database defines the mapping between Modbus registers and records, interpreting and giving meaning to the raw data values contained in the registers. Regardless of which peripherals are connected to the PLC, no changes to the drivers are needed.

But the Modbus protocol is just one possibility, chosen as an example because it is so widely known and used. As with all devices and protocols, there are many generic ones that are niche. Yet even for niche protocols, it makes sense to implement a module as generic as the Modbus module. A protocol may only be used at a single facility, yet that may still mean that it is used in many different setups (e.g. with different peripherals).

At Cosylab, we have helped with EPICS device support for such generic devices many times now. We felt the pain of doing the same kind of work several times. We have thus decided to develop a C++ base class which encapsulates the common functionality that needs to be implemented by any generic device support code; most importantly, this includes generating per-record handles dynamically based on information provided in EPICS records. This C++ base class builds on top of `asynPortDriver`, allowing us to reuse the basic functionality provided by this `asyn` class, and, crucially, to reuse the `asyn` per-record device support layer. We found this module, called `autoparamDriver` [7], to be generally useful, and are releasing it to the community.

* jure.varlec@cosylab.com

ISBN: 978-3-95450-237-0 ISSN: 2673-5512 doi:10.18429/JACoW-PCaPAC2022-FR012

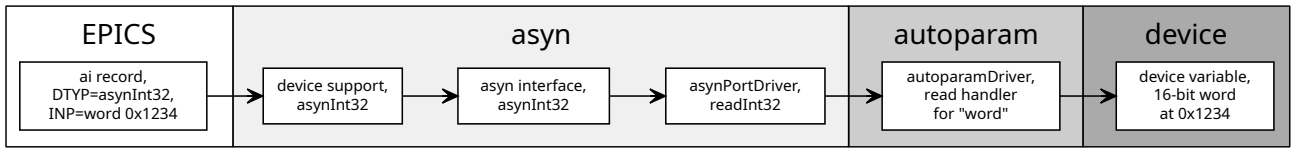

Figure 1: An example of reading from a device, showing the layers involved, starting from an EPICS record. Note how the DTYP field of the record specifies which `asyn` calls will be involved, while the INP field provides data for `autoparamDriver`. Specifically, `word` refers to a device function that returns a 16-bit word; `autoparamDriver` uses this in its implementation of `asynPortDriver::readInt32()` to find a read handler registered by a subclass. The other part of INP, `0x1234`, is used by the handler for `word` as an address on the device.

GOALS

`autoparamDriver` has several goals, listed here in order of priority.

1. *Allow the EPICS database to specify the mapping between process variables and device variables* (e.g. device registers) without modifying the driver or device support code.

2. *Provide approachable documentation.* `asyn` documentation, is quite general and not very EPICS-oriented.[1] The intent is to complement it with a gentler introduction to the concepts involved in writing a driver.

3. *Reuse as many of `asyn`'s facilities as possible and sensible.* The intention is to save development time[2] at the cost of conforming to certain constraints imposed by the `asynPortDriver` base class.

4. *Support different paradigms of processing `I/O Intr` records.* Some examples: processing based on hardware interrupts, from a periodic scan thread in the driver, and on writing to a register from another record. `asyn` does a good job at this and `autoparamDriver` should make this easier if possible, but not more difficult.

5. Related to the goal of providing approachable documentation, *provide a more ergonomic API than bare `asynPortDriver` when possible*. Internal details of `asyn` leak through its APIs, along with concepts that are hard to grasp or are heavily overloaded. A good example of both is the concept of "reason": reasons of different kinds are passed to `asynPortDriver` functions as parameters, but can mean different things depending on context. This can be difficult to follow even for an experienced developer.

DESIGN

The `autoparamDriver` EPICS module provides the `Autoparam::Driver` class, which is derived from `asynPortDriver`, and, in turn, needs to be subclassed and extended with functionality needed to talk to a specific type of device. There are two ways to look at such a driver: the end-user point of view and the driver developer point of view.

From the point of view of the end user, who is tasked with setting up an EPICS input-output controller (IOC) and its database for their particular situation, it all starts with the EPICS database (see goal 1 above). The database provides the information needed to transfer data to or from the device, and the layers underneath take care of it. This is shown in Fig. 1 where a "word" at address 0x1234 is requested from the device. The user needs to know

- which `asyn` device support layer is needed,
- which "device function" they want (e.g. `word`),
- and which parameters this function needs (e.g. `0x1234`).

That a particular device support needs to be chosen explicitly may seem superfluous, especially because `autoparamDriver` ties it to the device function (as we will see later). However, this is how EPICS records work, and for good reason.[3]

The driver developer, on the other hand, has much more freedom in how to approach the problem. There are three issues to tackle:

- opening "channels" or "handles" to the device based on which records are instantiated in the database,
- handling read and write requests from records,
- and pushing data to `I/O Intr` records, i.e. records that are to be updated in response to events originating on the device.

In the rest of this section, we will take a look at how `autoparamDriver` approaches them and what needs to be provided by the subclass of `Autoparam::Driver` and its ancillary classes to make things work. To make things easier to follow, the APPENDIX has an abbreviated listing of the `Driver` class.

[1] Excepting, naturally, the chapter on EPICS device support.

[2] This goal had high priority during the development of `autoparamDriver` because other in-development drivers depended on it. Its importance will diminish with time.

[3] A record can change its behavior significantly based on which device support is in use. For example, `asyn` allows an `ai` record to be used both with `asynInt32` and `asynFloat64` interfaces which handle unit conversions differently.

ISBN: 978-3-95450-237-0 ISSN: 2673-5512 doi:10.18429/JACoW-PCaPAC2022-FR012

Device Variables

When an EPICS records that uses `asyn` device support is initialized, it registers a handle for itself by making a call that ends up executing `asynPortDriver::drvUserCreate()`. This function is overridden by `autoparamDriver` and is the point where handles to device data are instantiated. These handles are called *device variables* as that is what they represent. Each variable has an associated *device address*. Based on the string provided in the record's INP field, which is passed to `drvUserCreate()`, a variable is created in three steps:

1. The first (and possibly only) word of the input string is taken as a *device function*. We only proceed if a read and/or write handler has been registered for this function (as described later).

2. The rest of the input string is passed as-is to the function `parseDeviceAddress()` which is implemented by the subclassed driver. This function returns a `DeviceAddress` object.

3. If this address was encountered before,[4] the existing `DeviceVariable` object is returned. Otherwise, the `createDeviceVariable()` function implemented by the subclassed driver is called to create a new object.

The `DeviceAddress` and `DeviceVariable` are abstract classes. They are meant to be subclassed so that they contain data relevant to the device.

Handlers

`Autoparam::Driver` overrides `asynPortDriver`'s read and write functions. These functions receive an `asynUser` pointer as a handle to the record that called them, and this handle contains a "reason" which can be used to determine which data is requested. Thus, an implementation of a read or write function typically contains a `switch` statement or an `if-else` ladder to dispatch on the "reason".

`autoparamDriver` provides that level of dispatch itself by recognizing that generic drivers commonly refer to a *device function*, such as the `word` function in the example shown in Fig. 1. `autoparamDriver` will examine the provided `asynUser` handle to find the associated `DeviceVariable`, then it will find the handler registered for the function associated with the `DeviceVariable` and call it. The subclassed driver thus needs to implement handlers for each device function it knows, and register these handlers in its constructor.

Read and write handlers are static functions[5] that take a `DeviceVariable` reference as an argument. They can cast it to whichever subclass the driver actually uses, thus getting access to everything the handler needs to communicate with the device. As an example, the signature of a read handler for 16-bit words looks like this:

```
Result<epicsInt32> readWord(DeviceVariable &var);
```

Note that the value read from the device is returned as part of the `Result`. This object also contains the error status and, if required, can override the alarm status and severity of the EPICS record that made the call. The type of data that it passes to `asyn` is given in its template argument. There is no mention of the 16-bit data type that is used to talk to the device: that is an implementation detail of the subclassed driver.

When the subclassed driver registers a handler that uses `epicsInt32` data type on the `asyn` side, the `Autoparam::Driver` base class knows that the data will move through the `asynInt32` interface and that this handler is a candidate when `asynPortDriver::readInt32()` is called. This is an example of a pattern that pervades `autoparamDriver`: instead of specifying the interfaces with an `enum` or having them as part of function names as is normally done in `asyn`, they are implied by the data types used. Scalars are passed as `epicsInt32` or `epicsFloat64`, digital IO uses `epicsUInt32`, arrays are wrapped in `Array<T>` and strings are wrapped in `Octet` (which is an `Array<char>`). There is a 1:1 mapping between data types and `asyn` interfaces.

Interrupts

`I/O Intr` records are processed via the mechanism provided by `asynPortDriver`, which is built on top of *asyn parameters*. Each `DeviceVariable` is backed by an `asyn` parameter, which is dynamically created during the call to `drvUserCreate()`; this is where the name of `autoparamDriver` comes from.[6] This means that processing `I/O Intr` records is done the same way as with plain `asynPortDriver`:

- set one or more scalars using `setParam()`, then call `asynPortDriver::callParamCallbacks()`;
- for arrays, use the `doCallbacksArray()`.

For scalars, `asyn` parameters take care of determining whether the value of a parameter has been changed and whether (and which) records needs to be processed.

`setParam()` and `doCallbacksArray()` are not *quite* the same as `asynPortDriver`'s `set*Param()` and `doCallbacks*Array()` family of functions. First, they take a reference to a `DeviceVariable` as the first argument. Second, they are *templates*. This continues the pattern noted earlier: `autoparamDriver` uses data types to determine which `asyn` interface to use. Thus, passing an `epicsInt32` to `setParam()` will dispatch to `asynPortDriver::setIntegerParam()` and the programmer does not need to deal with `asyn`'s naming idiosyncrasies (goal 5).

[4] Several records can refer to the same device variable.

[5] One of the constraints for `autoparamDriver` was that it has to conform to the C++03 standard, which constrains the design significantly, but allows for wider adoption.

[6] In truth, the name will stay appropriate even if this mechanism is replaced in the future, the basic design will remain the same. And it may well be replaced: reusing `asyn` parameters is in line with goal 3, but constrains `autoparamDriver` so that each device function can only be bound to a *single* interface. This was found to not be very limiting, but it may still be desireable to lift this restriction at some point.

ISBN: 978-3-95450-237-0 ISSN: 2673-5512 doi:10.18429/JACoW-PCaPAC2022-FR012

In accordance with goal 4, `autoparamDriver` allows one to process `I/O Intr` records

- during or after running write or read handlers,
- in response to hardware interrupts (e.g. from a callback function),
- or at any other time, in particular from a background scanning thread.

For each of these cases, there is a feature that supports it.

The code that calls read and write handlers can optionally also process `I/O Intr` records that are bound to the same device variable. A handler decides whether this is desired by setting a field in the `Result` structure it returns.

For a background scanning thread, it may be convenient to know which device variables have `I/O Intr` records that are interested in updates in a given moment. The `getInterruptVariables()` function returns a list of such variables.

For some devices, hardware interrupts need to be explicitly enabled, or a subscription needs to be set up for each device variable of interest. When a driver is registering handlers, it may also register an *interrupt registrar* function. That function is called (a) when the first record's SCAN field is switched to `I/O Intr`; (b) when the last record is switched away from `I/O Intr`. In other words, no matter how many records are bound to a particular device variable and have SCAN set to `I/O Intr`, only one call to the registrar will be done. The registrar function can then set up (or tear down) the subscription to hardware interrupts.

CONCLUSION

Let us summarize how the design of `autoparamDriver` supports its goals.

The primary objective is allowing the driver author to create a generic driver; such a driver facilitates binding device variables to records without modifying and recompiling the driver. To this end, `autoparamDriver` provides a mechanism to instantiate `DeviceVariable` objects that the subclassed driver can use as it sees fit. Parsing of device addresses that are entered into the EPICS records is left to the subclassed driver, allowing lots of freedom. Only the first word is interpreted as a *device function*, allowing the subclassed driver to be designed in terms of *function handlers*. This reduces the amount of necessary boilerplate.

`autoparamDriver` relies on the `asynPortDriver`'s *parameters* to help with processing interrupts, which constrains the design a bit, but builds on top of a well-tested foundation. `asynPortDriver`'s facilities also allow interrupts to be processed in a number of different scenarios.

While we recognize that API design is, to an extent, a matter of taste, we believe that `autoparamDriver` provides a more ergonomic facade over `asynPortDriver`'s functions by using templates instead of separate functions. The use of handlers instead of overloading read and write functions also furthers this goal. More importantly, `autoparamDriver` eschews various "reasons" that `asyn` uses, using instead `DeviceVariable` objects as handles to device data.

As for documentation [7], we aim to have not only a complete reference document, but also a gentle introduction to the concepts involved. Importantly, we do *not* provide an example driver. Instead, we provide a *tutorial*, each step accompanied with a short discussion of what needs to be considered. We hope that this will reduce cargo-culting and increase the quality of drivers based on `autoparamDriver`.

APPENDIX

Abbreviated[7] public interface of `Autoparam::Driver`.

```
namespace Autoparam {
class Driver : public asynPortDriver {
public:
    explicit Driver(const char *portName, DriverOpts const &params);
    virtual ~Driver();

protected:
    virtual DeviceAddress *parseDeviceAddress(std::string const &function,
                                              std::string const &arguments) = 0;

    virtual DeviceVariable *createDeviceVariable(DeviceVariable *baseVar) = 0;

    template <typename T>
    void registerHandlers(std::string const &function,
                          typename Handlers<T>::ReadHandler reader,
                          typename Handlers<T>::WriteHandler writer,
                          InterruptRegistrar intrRegistrar);

    template <typename T>
    asynStatus doCallbacksArray(DeviceVariable const &var, Array<T> &value,
                                asynStatus status = asynSuccess,
                                int alarmStatus = epicsAlarmNone,
                                int alarmSeverity = epicsSevNone);

    template <typename T>
    asynStatus setParam(DeviceVariable const &var, T value,
                        asynStatus status = asynSuccess,
                        int alarmStatus = epicsAlarmNone,
                        int alarmSeverity = epicsSevNone);

    std::vector<DeviceVariable *> getAllVariables();

    std::vector<DeviceVariable *> getInterruptVariables();
};
}
```

REFERENCES

[1] *EPICS database concepts*, retrieved August 2022. https://docs.epics-controls.org/en/latest/guides/EPICS_Process_Database_Concepts.html

[2] *EPICS hardware support database*, retrieved August 2022. https://epics-controls.org/resources-and-support/modules/hardware-support/

[3] *EPICS application developer's guide*, retrieved August 2022. https://docs.epics-controls.org/en/latest/appdevguide/AppDevGuide.html

[4] *asynDriver*, retrieved August 2022. https://epics-modules.github.io/master/asyn/

[5] *AreaDetector*, retrieved August 2022. https://areadetector.github.io/

[6] *The modbus module*, retrieved August 2022. https://epics-modbus.readthedocs.io/

[7] *The autoparamDriver documentation*, retrieved September 2022. https://epics.cosylab.com/documentation/autoparamDriver/

[7] Functions from the part of the public interface of `asynPortDriver` that are used by `asyn` must be public, but are not intended to be overriden further and are ommited here. See main text for the description and example signature of a handler, and short descriptions of some of the ancillary classes.