EPICS TOOLS ENHANCEMENTS AND TRANSPORTABILITY*

M. Bickley, J. Chen, C. Larrieu†, Thomas Jefferson National Accelerator Facility, Newport News, VA

Abstract

The Jefferson Laboratory electron accelerator is controlled by the *Experimental Physics and Industrial Control System* (EPICS), which was initially developed by the Los Alamos and Argonne National Laboratories, and which has since become an extensive collaboration among scientific institutions worldwide. In keeping with the spirit of cooperation and exchange fostered by the EPICS community, the Controls Software group at Jefferson Laboratory aims to produce portable software tools useful not only locally, but also at any EPICS site, and even at non-EPICS sites where feasible. To achieve this goal, the group practices several software engineering principles which have demonstrated success in producing sharable software. This paper first discusses those principles along with the practicalities involved in pursuing them, and then illustrates how they prevail within three different frameworks: the architecture and operating system (OS) portability provided by the EPICS environment, which assists in porting to other EPICS sites; the control system portability inherent in the Common Device (CDEV) abstraction layer, which facilitates porting to any supported control system; and the general system portability which follows from careful code design.

1 GENERAL CONSIDERATIONS

Because a control system is, by intent, uniquely customized to perform site-specific tasks, designing control system software for effective use at various unrelated sites might seem, at first, to require an amount of work disproportionate to the rewards such an endeavor might garner: not only must the programmer attempting do so solve the immediate problem which necessitates his application, but he must also devise some technique for dealing with the differences which arise among sites. Furthermore, he must decide how portable his application really can and should be before beginning to work on it.

Fortunately, these latter quandaries are substantially mitigated by the nature of the undertaking in question. It may be of a sort which is "algorithmically" portable from one site of a particular type, to another of the same type. Two particle accelerators, for example, might require software to measure and tune the phase of an RF cavity. While these sites may not use the same control system, the general solution to their common problem may be sufficiently parameterizable such that the same program could be used at both, if it were capable of controlling each system in the same generic way.

* Work supported by the U.S. Department of Energy, contract DE-AC05-84ER40150

† Email: larrieu@jlab.org

Another type of inherently portable application is the sort which provides basic facilities closely tied to the system being controlled. Examples include data acquisition, logging, and retrieval systems, human interfaces for monitoring and managing the underlying physical system, and utilities for measuring and analyzing the system in a user-specified manner. Such tools are fairly useful in a general sense across control systems and physical machines, so that for every site to design its own set seems tantamount to "reinventing the wheel". Ideally, the site-specific information should be configured via some localization method.

Of course, there are good and bad reasons for one site to avoid reusing software from another. In order to create a general purpose tool which another site will want to use, the designer must in some way address both. Among the former category are considerations arising from concern over efficiency and compatibility. Does the application make assumptions about the underlying system such that it works well in one situation but not in another? If it generates output data, will it do so in a format which other tools can read? Notorious among the bad reasons is reluctance arising from the well-known "not developed here" syndrome, a psychological phenomenon pervasive in the engineering world. It derives, perhaps, from the fear of becoming dependent upon the work of someone whose interests may not necessarily coincide with one's own.

This leads to a consideration of those factors which might induce a developer to produce code specifically designed only for local use. The benefit of such an approach is that it may allow for a tight integration among the tools in a local suite: if an application can make assumptions about the existence of other facilities, it can incorporate their features, providing users with multiple points of entry to the various system tools. Additionally, designing for a known system allows for faster development –because the programmer knows exactly the environment in which his application will run, he can avoid having to handle certain special exceptions which might arise from one site to another. Furthermore, a small user base will likely limit requirements and feature creep.

Note, however, that these assumptions about the environment imply that the benefits pertain only if the situation for which the application is written persists over time. What happens if something about it changes? What if one of the subsystems upon which the code depends is removed, or, more drastically, if the control system changes (not so preposterous: the CEBAF accelerator was switched from TACL[1] to EPICS during the commissioning of the machine in the mid '90s). The programmer can ameliorate the problems arising from such occurrences by designing his application to be portable from the outset. *Fundamentally,*

0-7803-5573-3/99/$10.00@1999 IEEE.

the principles pertinent to designing portable code are also those which lead to reliable, maintainable, and extensible code.

While the primary mission of the controls software group at Jefferson Lab is to support the operation of the accelerator, we find that by designing our code to be portable, we are investing a small amount of additional time and effort in the initial development phase in order to facilitate future development work.

2 EXTENDING AN EPICS APPLICATION

The EPICS [2] system provides a fairly simple framework for developing control system client applications, by providing site-specific configuration files and a C/C++ application programming interface (API). All high-level interaction with the control system can be conducted through the channel access library, which is a set of routines for establishing connections to named parameters in the control system. Within this framework, the task of designing software for porting to other EPICS installations reduces to producing code based on UNIX portability standards to address the issue of OS and machine independence, and modularizing those portions which make use of site-specific subsystems.

One useful application which was originally written at APS, then subsequently enhanced and modularized at Jefferson Lab, is a real-time data plotting utility called StripTool. While the original version was designed specifically for EPICS, the revision produced by Jefferson Lab was designed with the intent that it be control system independent. Because its functionality was initially quite simple in concept (the user just supplies the name of a control system parameter whose value he wishes to see plotted), the process of compartmentalizing the platform-specific code was relatively straightforward: all functionality was broken out into modules, so that the data acquisition, data buffering, graphical presentation, user interface, and timing components all belonged in separate modules with very well defined interfaces.

The process of transporting StripTool from one EPICS system to another is trivial, because it capitalizes on the EPICS-supplied configuration files and compilation instructions ("makefiles"). The interested party can, for all intents and purposes, simply download the source code, run the EPICS build command to create the binary executable, and then run it. In order to port StripTool to another control system, however, the local programmer must supply a "plug-in" for the data acquisition module, which amounts to writing a minimal set of routines as specified in the *StripDAQ* file, and linking them into the compiled program. It took just a couple hours to implement a CDEV data acquisition module, following this procedure. Now, both the Channel Access and CDEV modules are included with the source code, and the proper one is linked depending upon the build environment.

After these initial enhancements, the natural progression was to provide some support for viewing old data in addition to that which had been acquired through the real-time acquisition unit. Because StripTool only buffers the most recent data for some selectable time span, however, it requires some other facility from which to acquire the older data. In other words, providing such functionality introduces a dependency upon an auxiliary sub-system. Because a data logging and retrieval system is pretty much essential to any control system environment, several of the sites using StripTool have developed their own custom applications to serve this purpose. In order to accommodate all sites interested in incorporating this functionality into StripTool, another module, *StripHISTORY*, was defined. It describes a minimal set of routines specifying only those features of the system which StripTool requires. Regardless of how sophisticated a particular site's archiving service may be, StripTool just requires 3 functions in addition to an initialization routine: (1) "get data for parameter X over time range (t_0, t_1)", (2) "exchange this data for a new time range", (3) "free this data". By implementing just these three routines as the "glue" between the application and a specific subsystem, a local programmer is able to hook his unique archiving service into StripTool with only a minimal amount of work, while still ensuring that the resulting code need not be merged back into a central repository. This last point is important, because it helps to preclude situations which lead to initially diverging, then redundant parallel work, brought about by small changes in code from one site to another which never synchronize.

This technique is, in some sense, a "reverse black box" (it's also the model used for operating system device drivers), in that the application defines an empty plug-in mechanism, which can be filled by site programmers to translate an application's requests for auxiliary services into a protocol which the underlying subsystem can understand. However, if a site does not wish to use the specific feature, it need take no special action. This approach allows for integrating system tools by defining the interfaces between them in simple terms. A concomitant benefit of this approach is that, as the subsystems become more integrated, the interfaces will necessarily become more detailed, leading to better documented code.

3 THE CDEV ABSTRACTION LAYER

The Common Device API (CDEV) is an abstraction layer developed at Jefferson Laboratory[3], which allows the various subsystems of a control system to be addressed in a generic and consistent manner. It has been used to integrate multiple control systems, as well as to incorporate client-side applications into the control system.

An application designed specifically for CDEV must adhere to conventional portability standards to remain portable across platforms, and becomes entirely dependent upon being run in a special environment, which the local programmer must build, install, and configure. While this process may not seem easier than simply writing some

small amount of "plug-in" code, it does present the benefit that the requisite work represents a one-time investment, whereas having to write small bits of code for lots of applications can become wearisome, particularly when those bits are intended to perform the same task!

An EPICS Channel Access component has been written which allows CDEV applications to talk to Channel Access without directly relying upon the EPICS code. This has the effect of moving the control system dependency away from the user application and into the intermediary layer. Several EPICS utilities, like the alarm handler and medm have been modified to use CDEV in lieu of Channel Access, allowing them to be used in conjunction with non-EPICS control systems.

The controls software group at Jefferson Lab have also developed several new utilties. Zplot is a motif application that plots device attribute values against their coordinates along an accelerator. Xtract, the "X-windows Tool for Recording And Correlating Things" allows the user to change the system in a highly configurable manner, measuring various parameters along the way, resulting in data describing a discrete function of the stepped parameters. In this regard it is a general purpose experimentation program.

4 PORTABILITY THROUGH OBJECT-ORIENTED DESIGN

One of the greatest benefits of object oriented programming language like C++ is that it provides support for designing abstraction into an application. In designing a new application, the desired end product can be conceptualized as a virtual machine, comprised of distinct parts. The programmer's task is to forge actual software constructs (*Classes*, in C++ OO terminology) from these general specifications. The interesting thing about this process, is that it leads to a natural separation between the conceptual nature of the components and their corresponding implementation.

At Jefferson Lab, we use the EPICS archiver to log tens of thousands of control system parameters continuously, producing massive amounts of data which are subsequently compressed and cataloged in a locally developed database. Of course, once the data is nicely organized, one requires some mechanism by which to retrieve it: a program to facilitate browsing through the archived data, and converting it into a useful format. Because such a tool would be generally useful to other sites, and because we anticipated that our archiving system will change as we phase in new subsystems, we decided to develop a portable system based upon OO design principles.

The resulting tool (XARR: the Xwindows ARchive Retriever), as used at Jefferson Lab, is comprised of 3 layers. At the top is the graphical interface, which is implemented in C++, using the Motif widget set. Beneath this is the database layer which catalogues the available data. At the lowest level is an I/O library for reading and writing data from and to the storage medium. Because the two bottom layers are primarily specific to Jefferson Lab's storage system, while the top layer is not, the interface between them is implemented as a set of three abstract object types, corresponding conceptually to the components describing a general purpose data retrieval machine and the items it would require to perform its task: a DataSource, a DataHandle, and a DataBuffer. The DataSource enumerates the objects stored in the archive, provides lookup and search capability, and retrieves data. The DataHandle conceptually represents a way to identify some unique parameter in the archive. The DataBuffer encapsulates the data for a particular DataHandle over some time range, and provides methods for iterating over the contained data points.

The programmer interested in building XARR for use with some other archive must create three C++ classes derived from those named above, overriding the default behavior with site-specific details. When the application is compiled, the appropriate class definitions are included in the main startup routine.

5 CONCLUSIONS

While the ultimate goal in portability is to eliminate as much site-specific "tweaking" as possible, this often proves unattainable without also decreasing the efficiency or usefulness of an application. The most useful sort of portability within an open-source environment is the kind that allows local programmers at various sites to acquire code which they can easily modify to suit their own needs. Fortunately, this naturally follows from good design and coding principles. As illustrated above, by designing code modularly, those portions which require modification from site to site are isolated and readily found by the local programmers. If the code is also well documented, then the local programmer may more easily and confidently supply the necessary site-specific modifications, helping to relieve the "not developed here" syndrome. Because the local code is isolated in distinct modules, the process of upgrading to new versions of the software reduces to plugging in those site-specific modules to the new source code. While all these good things benefit the recipients of the "free" work, they also benefit the original developer and his site by providing software which is easily maintainable, extensible, and likely to remain useful through system changes.

6 REFERENCES

[1] K. S. White, H. Shoaee, W. A. Watson, M. Wise, "The Migration of the CEBAF Accelerator Control System from TACL to EPICS", *CEBAF Controls System Review*, 1994, Newport News, VA.

[2] Leo R. Dalesio, et. al., "The Experimental Physics and Industrial Control System Architecture: Past, Present, and Future", *International Conference on Accelerator and Large Experimental Physics Control Systems*, Oct. 1993.

[3] J. Chen, G. Heyes, W. Akers, D. Wu and W. Watson III, "CDEV: An Object-Oriented Class Library for Developing Device Control Applications", *Proceedings of ICALEPCS 1995*, p 97.