CERN/ECP 94-13
10 October 1994

USE OF OBJECT-ORIENTED TECHNIQUES IN A BEAM-LINE CONTROL SYSTEM

D.R. Myers and W. von Rüden
ECP Division, 1211 Geneva 23, Switzerland.

H. Butler
Los Alamos, U.S.A.

J. Yang
University of Science and Technology, Hefei, China.

Abstract

We describe the use of object-oriented programming in the control data-acquisition system for the upgraded CERN neutrino beam-line. C++ in conjunction with Posix threads running under Lynx-OS have been used in several front-end PCs. These communicate using Remote Procedure Calls over ethernet with a workstation running the supervisory package, FactoryLink.

Presented at *Computing In High Energy Physics*, San Francisco, April 21-27, 1994.

Use of object-oriented techniques in a beam-line control system

D.R. Myers and W. von Rüden,
ECP Division, 1211 Geneva 23, Switzerland.
H. Butler,
Los Alamos, U.S.A.
J. Yang,
University of Science and Technology, Hefei, China.

Abstract

We describe the use of object-oriented programming in the control and data-acquisition system for the upgraded CERN neutrino beam-line. C++ in conjunction with Posix threads running under Lynx-OS have been used in several front-end PCs. These communicate using Remote Procedure Calls over ethernet with a workstation running the commercial supervisory package, FactoryLink.

1. The WANF Controls Project

Two experiments at CERN, CHORUS and NOMAD, are using the rejuvenated West Area Neutrino Facility (WANF) and the original beam-line control system, based on now obsolete machines, has had to be replaced [1]. In order to achieve this an industrial control package, FactoryLink [2], has been installed on a powerful HP workstation and interfaced to the existing CAMAC hardware via two front-end machines, one for beam control and one for flux monitoring. The front-ends are industrial PCs running the LynxOS real-time UNIX system [3]. On both PCs an identical software *daemon* is installed which receives commands from the workstation and controls the CAMAC via a VIC bus. The daemon is written in an object-oriented fashion and is matched to the hardware configuration at run-time by instantiating the appropriate set of objects which have been defined in a Filemaker-Pro database [4].

FactoryLink consists of a real-time kernel to which is interfaced a wide range of off-the-shelf tasks for alarm handling, data logging, trending, timing and so forth. The kernel incorporates a database consisting of named *tags* which store primitive data types. Tasks may declare an interest in any set of tags and are informed if the value of a tag is changed.

A powerful graphic user interface, including the possibility for picture animation based on the value of a tag, may be constructed using an off-line drawing tool which interfaces to the screen via the X-Window protocol. Apart from an interpreter available for mathematical and logical algorithms, FactoryLink tasks are configured in a completely data-driven fashion. Although software exists to connect Programmed Logic Controllers, we were obliged to write tasks which interface to the front-end computers using our own protocol based on Remote Procedure Calls (RPCs).

The architecture of the system was fixed in the Autumn of 1992 and the implementation started in October of that year. A version of the complete system, which has been well received by physicists from the experiments, was available twelve months later. The initial manpower required was the equivalent of three man-years, with perhaps another eight man-months added for improvements. We believe that this figure is extremely low compared with the effort which would have been required using tradi-

tional methods to produce a system of comparable sophistication.

2. Class Design

Within each front-end PC there is a process running which is the recipient of RPC calls from the FactoryLink interface task. The calls contain *messages* for particular target objects specifying what *action* should be performed by the object and containing any necessary data. Thus, as an example, a message might be sent to a particular magnet power supply object requesting it to set a current of a certain value.

The system has classes which can be categorized into three different groups:

C-1 Specialized classes which correspond to actual WANF beam-line components.

C-2 Classes which model CAMAC branches, crates and modules, such as input registers or ADCs. These are classes dealing with the interface between the equipment and the computer.

C-3 Finally there are classes which also deal with the computer interface but which model the functionality at an abstract level. Classes in this group include binary input and output bits, analogue sensors, and so forth.

For the first two classes the strategy adopted has been to map software objects to real-world objects as closely as possible. It is interesting to note that failure to do this in a few cases led to problems which later had to be corrected. The utility of the last category is a technical one and these classes turned out to be more useful than might at first have been imagined. This is best illustrated by an example.

In order to switch the polarity of the beam-line it is necessary momentarily to close a switch. Thus, a Polarity Controller object has a pointer to an `OutBit` object which controls the switch, but it does not need to know if this is implemented in CAMAC, VME or anything else: the important thing is that an `OutBit` abstracts all the properties of a generic switch. In fact an `OutBit` object contains a pointer to a hardware module and an index number in case the module contains several bits. For C++, the implementation language, it is possible to define the type of a pointer so that if it is supposed to point to an `OutBitModule`, a module containing output bits, one could never load it with a pointer to a module containing ADCs, for example. Thus, the system is type safe and also run-time configurable, because the pointer may be loaded with the address of any class of object which inherits from `OutBitModule`.

It is in the class implementing the hardware module where the real work takes place. However, it is interesting to note that the actual code executed will be selected only at run time. In object-oriented terminology this feature is called *polymorphism*. For example, all classes inheriting from `OutBitModule` must provide a function called `writeBit(index)`. The actual version executed is selected depending on the class of the particular object pointed to. If a hardware module is purchased from another manufacturer then a new class would be written to support it, but no other changes are needed anywhere in the system.

3. Implementation Issues

3.1 Threads

As a consequence of using objects which, at the current state-of-the-art, cannot easily be distributed over several processes, the problem arose as to how to handle asynchronous events. This has been solved by using Posix Threads [5] which are available under LynxOS. Threads are essentially light-weight processes running within the same address space. As an example, if an object decides it must wait for an interrupt it spawns a new thread to which control is transferred whilst the interrupt is pending. Meanwhile, the code in the initial thread can continue with something else.

Unfortunately Posix threads know nothing about objects but, by a subterfuge, it is still possible to transfer control to an object member function. Of course, if code in several threads needs to access the same variable then this must be protected by using mutual exclusion semaphores (Mutexes). With these two provisos, threads were found to be an elegant and powerful facility.

3.2 Object-to-Tag Mapping

No equally elegant way was found to deal with the problem of mapping PC objects with complex behaviour onto FactoryLink tags. Tags support only the functions read and write, whereas objects have functions (often called *methods*) which can respond to all sorts of commands. To illustrate this, consider a magnet which can be switched on or off or be set to a particular field. This could be modelled by a class with methods `Switch(onOff)` and `SetField(value)`. However, as FactoryLink does not support complex data structures, the magnet has to be represented by two independent tags (one boolean and one real).

The solution adopted was to impose a convention on all tag names such that characters after a "$" indicate a command. Thus, in our example we would have the tags:

```
bendingMagnet$onOff, and
bendingMagnet$setField
```

The first part of the name is transmitted to the daemon where it is looked up in a hash table in order to find the identifier (address) of the corresponding object. At this point, instead of being able directly to call the correct function, it is necessary to pass via an intermediate stage which first has to find the correct destination based on the command field.

3.3 The DataBase

Whilst the front-end daemons have the capability to instantiate objects of any class, a mechanism was required actually to specify these objects. Although an object-oriented database would undoubtedly be a better long-term solution, we used Filemaker-Pro on a Macintosh which was fast and effective with an excellent user interface.

The database has an item corresponding to each object specifying its class, name and parameter set. From this it is straight forward to produce configuration files for both PCs as well as wiring lists for the technicians. Files are also produced for FactoryLink listing all the tags with which the PCs must communicate.

3.4 The Survey Class

One of the jobs of a control system is to monitor various parameters to ensure they stay within pre-defined limits. FactoryLink provides a task to do this, but the question arose as to how the information could be collected. The solution adopted was to provide a `Survey` class, instances of which maintain lists of objects to monitor.

When an asynchronous survey object is started it spawns a new thread and transfers control to a member function which alternately sleeps and then sends a message to all the objects on its list. This asks *Have-You-Changed?* Each object must know how to compare its current value (or values) with its previous state and then reports the changes.

3.5 Object Independence

A control system may have many objects which supply values read from different hardware sources. Let us say that these are read out by one of the `Survey` objects. A design feature of the model adopted was that each channel appears as an independent object which is addressed by name. The user need not even know how it is implemented and this makes changes to the system very easy; all one needs to do is to modify the database so that the abstract object becomes associated with a different module object of the same base class.

Now, what happens when several objects are surveyed which are situated in the same physical module? To discover if their values have changed separate read requests will be sent to the module. This might be merely a slight inefficiency if all the values could be obtained by a single hardware access. However, if the action of reading a value clears all the hardware registers, then only the first value obtained will be meaningful.

To solve this the survey process was split into two parts. When any object adds itself to a `Survey` the module in which it is situated is added to a unique list. When a survey is started each object in the list of modules is asked to refresh all its values and store them locally. When the objects to be surveyed later obtain their current values these are then available without a new hardware access. This scheme has a number of useful features in that it works with hardware which does a clear on read, it is efficient and, finally, it guarantees that all values from a single module are obtained simultaneously. The last feature may be essential for data which are produced by a unique event.

3.6 Multiple Inheritance

Multiple inheritance was found to be essential for several of our classes. For example, an `ADCModule` is both a CAMAC module as well as a set of ADC registers. Multiple inheritance provides type safety in this case and also forces concrete subclasses of `ADCModule` to implement the required virtual functions.

4. Conclusions

Even though the C++ compiler available under LynxOS was old and rather unreliable, the use of object-oriented techniques for this application has been very effective. About 50 classes, corresponding to more than 20k lines of code, have been written, debugged and documented in twelve months. Of these classes 20 are specific to the WANF beam-line, but the rest could be re-used in other applications.

Object-oriented code is indeed modular and it has been straight forward to add classes for new types of hardware. Adding objects for which classes already exist requires no coding changes whatsoever. It is even possible to add new objects without interrupting the daemon.

In summary, we have used commercial tools wherever possible and object-oriented programming for that part of the project for which a commercial solution was not available. This strategy has enabled us to build the control system on time and with a quality we could not otherwise have hoped to achieve with the time and man-power available.

5. References

1 H. Butler, D.R. Myers, W. von Rüden and J. Yang, *Beam-line Operation using an Industrial Control System and Distributed Object-Oriented Hardware Access*, Real Time'93 Conference, TRIUMF TRI-93-1, Vancouver, Canada, June 8-11, 1993, pp 180-183.

2 FactoryLink IV Overview, United States Data Corporation, 1551 Glenville Drive, Richardson, Texas 75801, U.S.A.

3 LynxOS User's Manual, Lynx Real-Time Systems, Inc., 16780 Lark Avenue, Los Gatos, California 95030, U.S.A.

4 FileMaker-Pro User's Guide, Claris Corporation, 5201 Patrick Henry Drive, Santa Clara, California 95052, U.S.A.

5 IEEE Draft Standard For Information Technology -POSIX- Part 1: System Application Program Interface - Amendment 2: Threads Extension, P1003.4a/d8, October 1993.