Comparative Analysis of the CERN Accelerator Control Systems

Chip Watson
26 January 1998

1. Introduction

This report is part of an ongoing analysis of the CERN PS and SL control systems aimed at a long term evolution into a single integrated system for operations during the LHC era.

The following material focuses primarily upon software, but addresses hardware to some extent, particularly when it has an impact upon the software design and capabilities. Also, this analysis is restricted to software which is unique and central to the control system, and excludes general purpose software such as word processing or spread-sheets. It is, of necessity, sketchy in many places, with more attention given to areas either where the differences are small (and convergence is easy), or where work in these areas, though difficult, is of high value to a more seamless operation of the entire complex. As only 3 months of analysis were done to prepare this work, there are still many areas of software which have not been examined, and certainly many additional areas for fruitful software re-use and sharing yet to be found. In particular, the area of high level physics applications is nearly untouched, as it is necessary to address other systems first in order to enable integration at such a level.

The discussion will flow from the lowest level of the control system (front ends, I/O), to the highest level (operator interfaces, some machine physics applications), with a separate section dealing with infrastructure which spans both levels (timing, alarms, error logging).

Ideas for future convergence of the two control systems are given in the various sections and summarized at the end of the report ("Prioritized List of Activities" on page 17). These represent the ideas of the author, and are intended as a starting point for discussions with the PS and SL divisions' controls groups.

2. Architecture

The overall architectures of the two control systems are similar in gross details. That is, both follow the *standard model* for accelerator controls in having (1) operator workstations or consoles for the operator interface and for high level and global controls, (2) a second tier of distributed computers for device access, typically running a real-time kernel and typically in a VME form factor, and finally (3) hardware interfaced via a fieldbus or VME cards, and sometimes containing embedded processors forming a third tier of computing.

In the specifics, however, the two systems are quite different, reflecting both their historical backgrounds and the development *cultures* which produced them. These differences are most notable in the server (middle layer) architecture, as well as in the use of databases. These differences will be brought out more fully in the following sections, but in order to put the next sections in context, it is worthwhile to first give an overview of these differences.

In the SL control system, there is a database of machine set points, the settings database, implemented as Oracle tables on a Unix host (console level). Many applications read and write these tables in order to manipulate the machine state (other processes transfer this control data to the front ends at an appropriate time). Similarly, there is a measurement database implemented in Oracle into which data collection processes write. In other words, the relational database is *between* the high level applications and the front ends. (Note that applications, for whatever reason, can bypass the Oracle tables).

This architecture was chosen for LEP in part to get a handle on the complex data management problem existing there due to the open architecture adopted for the front end software (few standards initially imposed upon developers).

In the PS control system, there is a real-time database implemented in each front end. Client applications write control values to, and read measurement data from, this database. On the surface, there are some similarities to SL here, but the differences are more significant. First of all, the database is not inherently persistent, nor is it relational. Secondly, it is accessed transparently through a client application programming interface (API) which does not present the control system as a database to the client program.

3. Front End Software

Front end software can cover a wide variety of topics, but for this report it will be defined as including at least supervisory control capabilities (read / write values to hardware), and a network server and communication protocol for access. It may also include data checking and alarming (on state or limits), data monitoring (without client polling), and local front end control algorithms. The client layer will be dealt with in the following section, “Equipment Access Libraries” on page 4.

3.1 Architectures

3.1.1 PS Architecture. As a general rule, the PS control system tends to be more database oriented for the front end software, spanning a configuration database (Oracle) on a host, and a real time database (data tables) in each front end. Conceptually, there is on the front end server a table per device type, a row per device, and a column per device property. For properties which are cycle dependent (PPM, or pulse to pulse modulation in PS terminology), there are multiple columns for a single property, one for each possible virtual machine.

Supervisory access to the hardware is implemented as read/write operations to columns of the records in that database. Typically, operations read or write to multiple rows or devices in a single call. For PPM properties, device type specific real-time tasks transfer data between the database and hardware periodically, on particular machine events such as the beginning of a cycle.

The client side accesses to the database are highly structured, with a separate function for each operation (read/write) on each property of each device type. These property access functions may be automatically generated if there are no special requirements. Strong type checking is imposed upon these functions through interface definitions contained in the host database (a function may be declared to use a property with read-only access, for example).

The real-time tasks may implement some form of surveillance, with the results placed into a device state property which will later be polled by an alarm client.

A data subscription mechanism which operates on a large number of devices and properties as a single entity is currently under development to re-implement a feature which existed in an earlier generation of the control system.

3.1.2 SL Architecture. The SL infrastructure and front end server is much more I/O connectivity oriented than the PS system. Network operations contain physical addressing information (instead of database references), and the server (message handler) deals with the request based upon this information. For some types of I/O (e.g. simple MIL 1553 connected devices), the server executes the request itself, immediately. For complex devices, or those on "unsupported" bus types, the request is passed to another task, an equipment server. A reference to the correct equipment server is in this case the "physical" address contained in the request. Argument validation and type checking are deferred to lower level software (either the equipment server or an embedded system on the 1553 bus), and are not imposed by the front end framework software. The SL server itself has no configuration information for each connected device (this information is instead maintained on the host).

There is currently no support for data subscription in the server. Also, there is no standardization of surveillance of device state, although there are libraries available for a real-time task to emit an alarm into an alarm system (discussed later in this report).

3.1.3 Architecture Summary. In effect, the front end servers for SL and PS span different amounts of functionality, with the SL network server being much lighter weight, serving only as a vehicle for location independent connectivity, whereas the PS server includes considerable data management and validation features. It should be noted that both are using the same remote procedure call (RPC) system, but with different remote procedures implemented.

This main difference between the two systems reflects the organization of the software development within the division. In the PS division, all low level software is developed by a small number of people within the controls group, whereas for SL division, hardware groups (and others) have implemented the embedded software or equipment servers which are responsible for much of the behaviour of the low level systems. The open architecture of the light weight server imposes fewer restrictions on this diverse set of developers, which is both an advantage and a disadvantage.

3.2 Low Level Device Drivers

For the most part, there is no sharing of device drivers, although discussions on this topic are beginning. No attempt has been made as part of this analysis to catalogue the drivers available in each system.

3.3 Convergence

Convergence of these two systems will be difficult in that there is much that is different between them. That said, most of the differences could be hidden behind a compatibility layer most likely implemented on the client side, at which point either both systems could be kept, or one system could be frozen and the other chosen for continued development. The next section on client application programming interfaces (API) adds additional information to this topic, so the discussion is continued there.

At present, SL is upgrading software for the SPS as part of a project also aimed at LHC operations, and this upgrade will most likely move the two server systems closer together in terms of functionality.

4. Equipment Access Libraries

The equipment access libraries provided by the 2 controls groups (PS Equipment Access, and SL-Equip) have a great many similarities in their most important functionality, namely in reading and writing control system values. The definitions of the API's have mostly the same arguments for supervisory I/O (albeit arranged differently). The most notable difference between the two is that the PS API is quite a bit larger in that it contains a considerable amount of support for generic applications, with list processing and extensive access to meta data (data associated with a value, such as units, or alarm limits). Some of this functionality for SL division is instead contained in another library created by the operations group.

4.1 Features and Capabilities Comparisons

It is helpful to compare the libraries on the basis of their capabilities and features, and not in the exact manner in which arguments are passed. To that end, the following discussion breaks the problem into 5 topics: (1) the "view" of a control system presented by the API, (2) performance features like asynchronous operations, buffering, and array operations, (3) data subscription capability, (4) supported data types, (5) standardized support for "generic" applications, and (6) other miscellaneous observations. This comparison deals mostly with the equipment access libraries, but some additional notes related to the operations library are included as well.

4.1.1 View. Each control system API organizes its methods around a view of the control system. It is at this level that the PS and SL libraries are most similar. Each treats the control system as a set of devices upon which operations may be performed. Devices are specified by a family name or number (class of device) and a member name or number. PS tends to use numbers more, presumably for performance, and provides mapping functions between name and number. SL must occasionally also specify the front end host name as family+member is not always unique.

The views are somewhat different in that PS further defines a device as a collection of properties against which 3 operations may be performed: read+write+dataless. SL instead only specifies that devices may be sent a 6 character "action", which is most often used to select a property, but this is not imposed. E.g. for the same action, read and write are not forced to be symmetric.

4.1.2 Performance Features. Neither PS nor SL support a truly asynchronous interface; all calls are complete when the call returns. (Data subscription capabilities are discussed below).

Both PS and SL support array type operations, i.e. operations on arrays of devices. In PS' case, all operations are in fact array operations, and the exact operation is specified partly by the method invoked (e.g. read()) and partly by the property code specified. SL has a more flexible interface in that the operation is specified entirely as an argument (the "action"), and the array call includes the specification of an array of actions. This allows mixed operations (operations on different properties, or in principle mixed read and write) to be performed in a single call.

While SL does not support an asynchronous calling interface, it does internally execute array calls to multiple servers in parallel, asynchronously waiting inside the library for the (partial) replies from each.

PS also does not have asynchronous calls, but it does support a mode in which calls are buffered, and executed later. The system is not truly asynchronous in that the work is not started until the flush call. For older front ends (Norsk Data), the control system buffers these multiple calls into a single network operation, achieving higher performance on these older systems. For the newer DSC's, this optimization has not yet been performed.

4.1.3 Subscription Capabilities. Neither PS nor SL have a general purpose data subscription capability that maintains the same view of the control system as a set of devices with properties. That is, it is not possible to subscribe to an arbitrary set of properties of an arbitrary set of devices.

The PS control system has a new data subscription mechanism now in the testing phase which allows subscription at the level of a large collection of data called a working set (see "Console Manager" on page 8) and does not allow subscription at the level of one property. This is optimal for some class of applications such as the working set displays, but forces other applications to deal with larger amounts of data than they may require.

Also within the PS system is an application, the *passerelle*, which serves as a gateway between PC client applications and the main PS control system. In addition to handing synchronous requests, this application can poll the control system for data, and send asynchronous updates to attached clients. On the client side it implements the same protocol as is used to control Isolde.

The SL system has a monitoring system built up from *monitoring black boxes*, applications which poll the control system for data, and then write that data to Oracle tables. Client programs may either read the data from the database, or may subscribe directly to the black boxes to obtain the data with lower latency. While this system is used extensively as an alternative to having each application directly poll the hardware, it is not flexible in that it is not possible to choose arbitrary sets of properties and devices for subscription. The set of data being monitored is fixed by configuration information for the black boxes, and is not under a client application's control.

4.1.4 Data Types. Both PS and SL support a subset of the primitive data types of C/C++. Neither supports unsigned short or unsigned long, and SL also does not support floats or long. Each supports arrays of the supported scalar types. In addition, SL supports mixed type structures described by an additional string argument. This ability to pass structures is also used for performance optimizations in that multiple simple data items may be packed into a structure for I/O in a single call.

Because the data transfer direction is specified as part of the function name, PS does not support bidirectional data transfers on a single call. SL does support bidirectional data transfers with some restrictions. For some calls, the data types must be the same in both directions. For calls using the string structure descriptions, the 2 structures (sent and received) may be different.

Data conversions are supported for all operations in PS (i.e. conversion between the server's implementation data type, and the client application's requested data type). SL defers this conversion to the lower level equipment server, which may not handle it correctly.

4.1.5 Generic Applications Support. Generic applications are those which are driven by an external file or database description. The control system API can assist in creating generic applications by managing some of the data needed to describe the control system to the application. SL has essentially no standardized support for generic applications built into the equipment access API. PS has a considerable amount. (For LEP, much of this data is available in Oracle tables, but without a uniform interface as in PS.)

For each property of a device, it is possible to obtain a number of items describing the data (property description, data type, read/write flag, continuous / discrete flag, field definitions for discrete data), plus a number of other items considered *dynamic* (in addition to value, units, upper and lower limits, resolution, tolerance, and *colour*, which reflects status and severity).

4.1.6 Miscellaneous. Both of the network layers for these packages are implemented on top of the SL RPC library, one notable highlight in software sharing between the 2 divisions.

A quick summary comparison of these 2 libraries, along with a comparison to the CDEV API (a device oriented API with a similar *view* of the control system), is included in Table 1.

4.2 SL operations library

While the SL-Equip package focuses primarily on data transport, there is another library of database routines within the SL system which provides an application with a good deal of information describing the configuration of the LEP controls. This library presents to some extent a different *view* of the control system, and in fact presents 3 views. One, a hardware view, is similar to the equipment access libraries. A second view, a logical view, provides logical devices (like magnet, instead of power supply). The third view, a physics view, allows an application to see the machine in terms of physics parameters like *tune*.

This library supports discovery operations such as obtaining lists of systems, sub-systems, and parameters for either hardware or physics systems (different functions operate on these 2 views, with different names for the categories for each). For the hardware view, the systems are analogous to device class and parameters are analogous to properties. For the physics view, it behaves as if there were meta-devices, like tune, having meta-properties, like horizontal and vertical tune, although this is not the nomenclature used.

Data type information, that is, the data type of an actual parameter which can be sent to hardware or obtained from hardware, is not as extensive as in PS equipment access. The data type information available only describes the communication between an application and a helper daemon, which itself has compiled in knowledge of the structures used in the various front end systems.

Other meta-data, such as valid values or limits, is available through another set of database routines.

4.3 Summary Observations on the API's

The SL equipment access API and implementation is significantly smaller and simpler than its PS counterpart in that it was designed to be only a location independent operation transport layer.

The PS API is rich in functionality and complexity, containing support for generic applications, and in many cases having more than one way to accomplish the same task. It is considerably more difficult to learn fully, and is tightly connected to the configuration database for the PS front end software.

The SL operations library is likewise rich in descriptive functionality, including both a physics and a standard controls view. It is not small, and makes extensive use of C structures to pass requests and results and hence is even more difficult to learn than the PS API.

4.4 API Convergence

Recommendations for convergence can be separated into different steps, each building upon the previous. Tasks from different steps may in some cases be tackled concurrently, depending up the ultimate goals of the effort and the resources available.

4.4.1 Basic convergence. At a minimum, there must be a common software interface to both SL and PS controls if there is to be any software sharing at the higher level. A basic common API should therefore be worked out between the 2 groups to include at least the following functionality:

1. get/set any named property of any named device
2. standardize at some level a status property for all devices (at least at the level of being able to ascertain if the device is useable and is providing valid data)
3. obtain for each property the following: value, limits, measurement precision, and units
4. provide data conversion to the application's preferred type
5. provide at least one performance feature (async calls, list calls, array calls)
6. provide some way of specifying timing information (like PS pls_line argument or SL event specification)

Both systems would then support explicitly the view that a device is a collection of properties whose values can be set or obtained.

This level of convergence would be sufficient to support a wide class of applications which maintain, externally to the equipment access library, all their own necessary additional configuration information. For example, a synoptic display program needs no further support as long as the display description is correct; i.e. it does not attempt to display a status register consisting of bit fields using an analogue meter. As another example, a steering program would need to have externally provided a list of measurement devices, correctors, and the desired orbit.

This level of convergence could be provided by providing wrappers around the existing libraries, including the operations libraries for the SL implementation of the wrappers.

4.4.2 Moderate convergence. A higher level of software sharing could be achieved if the equipment access layer and servers supported additional capabilities:

1. Publish/subscribe, otherwise known as monitoring.
2. "Discovery" capabilities, so that a program can learn, at run time, a list of devices, and the attributes/properties which a device supports, etc.

The first of these additions, *publish/subscribe*, is to enhance performance, especially for java applications, by filtering the amount of data a client must process. It would involve not just an API change, but also new capabilities for both control systems. With a careful design, much of the code for a monitoring system could be shared between the two systems. As is done with the *passerelle*, the new subscription service could serve as a gateway for new clients, and could itself be built on top of the existing subscription services or upon the existing read calls. Alternatively, all servers could be upgraded to support the new monitoring capability, and an optional gateway could be provided to reduce load on the front end servers.

Adding publish/subscribe also improves portability, by bringing the API up to the level of CDEV, with a view towards not only sharing between SL and PS, but also with CDEV sites. Most CDEV applications depend upon this capability.

4.4.3 Physics convergence. At this level, standardization would proceed to the point that the applications used at LEP which manipulate the machine from a physics view could be portable.

1. Property standardization. Standardization of beam sensors and magnetic elements (class names and property names) to the point that an optics application such as steering is portable between the two systems.
2. Machine description standardization, to the point that an optics application can discover a beam line layout sufficiently well that optical properties can be calculated.
3. Physics view standardization, with corresponding virtual devices analogous to what is currently possible in LEP controls.

4.4.4 Full Convergence. At this step, PS and SL would share not only an API, but the implementation of the configuration database, client libraries, and networking protocol. This would only be feasible once a common front end system was achieved. This final step may not be practical in the short run, as it would entail re-writing much of the front end software for the discarded system.

5. General Purpose Console (Client) Software

This category of software implements standard control features for an operator: data presentation, control input, and saving and restoring data values. It also includes the environment in which various applications are launched. Rather than just address a single topic at a time, the entire suite of these applications will be considered as a single entity. The alarm system, timing system, error system and physics applications will be treated separately.

One significant difference between the two control systems which tends to heavily influence all of the client software is SL's insertion of an Oracle (LEP) database between client programs and the front end servers. This control settings database is where an application writes a new value which is later propagated to the machine itself. There is also a measurement database from which data obtained from the hardware can be read. As a result of this architecture, many of the SL applications tend to be database applications, and not distributed or client-server applications communicating directly with the front end computers.

The first topic addressed in each section below is the set of applications used by operators to start other applications, control set points, and view measurements. Due to limited time, many custom measurement viewing applications, such as orbit displays, could not be covered.

A smaller discussion is also included on the topic of data logging and viewing of logged data.

5.1 PS

5.1.1 Console Manager. PS controls includes a special application (the console manager) which provides basic views of the control system, and is used to start other applications, as well as to manage the placement, iconization, and de-iconization of their windows.

One important aspect to understand about the PS console manager and applications is the notion of a machine (one of the several accelerators in the complex), and a context (the selection of a beam type and destination). Another key concept is the notion of working sets, which are lists of arrays of similar devices. A typical application in PS operates on a single machine, most often looking at one beam, and manipulating a small number of working sets specific to that machine.

The console manager, for example, runs attached to one physical machine, and can change beam selection via a context menu. Programs are started from the console manager associated with a single machine and beam. Three special displays are included in the “generic” support: (1) a tabular working set display, (2) a keyboard driven “knobs” window, and (3) an error viewer window.

The working set window (part of the console manager) displays a list of all device names (organized by type) in a particular working set, and a set of properties (typically a control value and a read-back or acquisition value). Reference values for these properties may also be displayed. There is a strong standardization as to how mismatches of various sorts are colour coded for display.

Users can transfer a (set of) device(s) to the knobs display, and interact with the control values for that device. The knob program runs as a separate (server) process. A special Motif thumb switch widget eases control operations. Multiple knobs can be displayed, and keyboard focus moved among them.

Specialized (non-generic) applications may be started from either the console manager or the working-set windows.

All configuration information for the console manager and associated working set and knobs windows is contained in Oracle tables. Valid beam (user) names, menu definitions, program lists for each machine, working set definitions, properties to display in the working set and knob windows, etc. are managed via the database.

This is a very well organized system, easily configured through the forms interface of Oracle. There is nothing in principle which would keep this system from being used on any other control system front end which had the notion that devices consisted of collections of properties which could be read/written.

5.1.2 Data Management. The following data management capabilities are included in the PS console manager: (1) compare control and readback values; (2) compare control to reference values; (3) copy reference to/from live system; (4) save values to an archive (like reference, but multiple slots; not for time history but rather to save a particular named setup); (5) log values to an ASCII file.

5.1.3 Data Logging / Trending. PS currently does no time based recording of values from the machine, except those captured by the “Log” capability of the console manager (i.e. to ascii files). It is also possible to capture machine data into Excel, which provides one environment for trend analysis (but not automated).

5.2 SL

5.2.1 Console Manager. The SL console manager is likewise presented as a menu bar, with cascading menus to start individual *tasks* (a task is one or more executables). The command line for a specific program includes window placement options so that programs can be positioned on a display in a convenient fashion. The menubar can also contain push buttons for frequently used operations (like “reset” of trips).

By convention, applications create their main windows with the same name as the *task* so that the console manager may control the windows through the X server, for example to iconize or de-iconize them, or even kill them (destroy the window). Applications may also be configured to be stopped through a configured signal number. In summary, this application duplicates the program start-up capabilities of the PS console manager, and includes similar window manipulations.

(Note: the windows created through the PS console manager are tagged with a particular context corresponding to a virtual machine or beam+machine selection, whereas the SL windows are associated with a particular task. Specifically, one selects a context, then starts the application, and the application may configure itself according to the context given to it. This once again shows the heavy influence of time slicing in the PS complex.)

5.2.2 Sequencer. One special application which can be started by the SL console manager is the sequencer. This is an application which likewise contains a menubar of cascading selections, where a leaf node does not start a single program, but instead executes a sequence of operations, each of which is a program invocation or function call. Output of spawned programs is displayed in a text window within the sequencer GUI. Sequences may be started from any node in the tree (executes all contained steps), and any node may be skipped, or may have a breakpoint (halt the sequence at that node).

This application is heavily used to execute, in a repeatable way, all of the steps necessary to prepare for injection, dump the beam, etc. The sequencer program is configured by text written in a special sequencer language which is then compiled into the sequencer program itself. Each line of this file represents a well defined operation on the machine. Any sequence is more easily modified than a typical program would be, and the “lines” of the sequence can in fact be complex programs.

5.2.3 Data Management. Two different general purpose windows into the control system are presented to an operator in order to edit the configuration of the machine. The first of these, the “trim” program, is used to edit the ramp functions for the cyclic part of the control system. The operator can choose a particular control category and parameter from menus, and (for the SPS) can graphically pick a section of the cycle from a supercycle display. Magnetic trims (edits) may be applied at the level of power supply current, bending strength, or at the level of machine tune (i.e. in a physics parameter space). The trim interface contains a powerful graphical editor, with an ability to change a single point or a time range up or down, the ability to add and remove points from the vector, and many other options.

Each edit of the trim results in an update to a database, and the history of the change is kept. The graphical interface is capable of overlaying any previous (historical) curve for comparison.

A second application allows an operator to manipulate a single time slice of the machine, one parameter at a time. Again, there is support for changing either low level parameters, or high level parameters.

Machine setup is captured in a “hyperrun” -- a database of all (time varying) machine settings. Each item has a history of edits associated with it as well as a most recent value, and these edits may be removed, and then compared to the most recent value. This rollback+compare capability is included in the trim program, for example. New hyperruns may be created by copying an existing one, or from an off-line model based calculation.

Whenever a parameter is changed, both of these programs update the Oracle settings database, not only for the changed parameter, but also any other coupled parameters. That is, the control database contains both the high and the low level parameter settings, so when one is changed, the other(s) must be as well. If one parameter effects many others, all must be changed.

Because it is the client application’s responsibility to effect all of these changes in an internally consistent manner, these applications tend to be more complex, and include a good deal of knowledge of the accelerator and its configuration database within them.

That said, there are re-useable pieces in this software. The data viewer used by the trim program is implemented as a separate program communicating with the trim program through shared memory. Also, there is a graphical timing cycle selector application which is also re-used by many SPS applications, and which also communicates with other applications via shared memory.

5.2.4 Data Logging / Trending. The LEP logging system is built upon a measurement system consisting of a single measurement server, plus multiple measurement black boxes each of which acquires data for one type of hardware, and writes the data to a measurement database. Client programs may obtain the data either directly from the black boxes via TCP/IP, or from the database.

The LEP logging system is a client of this measurement system, and logs data into Oracle tables in a ring-buffer fashion. The logging server monitors conditions and controls logging black boxes (on/off) based on the state of the accelerator. A clever design optimizes the storage requirements, and ameliorates the lower performance of using a relational database instead of a custom file format. A surveillance program checks data freshness and consistency (values in range), and generates an alarm if there is a problem.

There is a graphical user interface to the logging system which supports browsing the SL logging database, as well as the ST logging database. Not unlike the LEP Trim program, it presents a view of the parameter space as system, sub-system, sub-sub system, parameter, where the highest point in the parameter tree is the choice of data source (LEP, ST, etc.). One possible improvement in this approach is to have the view of the parameter space for logging be identical to the view of the parameter space for controls, namely device, property. Properties or devices which are not being logged could either (optionally) be greyed out or removed from the menu lists.

5.3 Convergence

5.3.1 Console Managers. There is a large functionality overlap between these two applications. A single one could be re-used if the notion of “context” were made abstract, and there was an option to switch between tracking via context and tracking via task. In the menu initialization, programs could be marked as to whether they were context dependent (beam/machine dependent). Some agreement would need to be reached on initialization via database or initialization via file, and the working set displays would probably be moved into a separate application. The view presented to operators would not need to be noticeably different from what exists today, and the amount of software maintained would be correspondingly reduced.

5.3.2 Sequencer. The SL sequencer could potentially be used as is, except for a few statements that directly execute operations instead of forking other programs. Because of the aperiodic, long lived nature of the AD machine is similar to LEP, this might be a useful tool for that project.

5.3.3 Data Management. The LEP software’s ability to deal with data at the level of physics operations, in a manner symmetric with hardware control (as presented to the operator), is a concept worth copying to the PS control system. Additional study would be required to make recommendations upon how to achieve this goal. One approach is to standardize at the equipment access API level all of the physics parameters into special “devices”, and an alternate approach is to copy the SL database design for physics parameter support to PS.

PS controls working set displays, and standardization of device status, would similarly be valuable both to the cycled SPS machine, and the aperiodic LEP machine. This would be simple to port if a more advanced API were available as recommended above (see “Moderate convergence” on page 7).

5.3.4 Data Logging / Trending. As there is little in place at PS at this time, convergence is straightforward in principle. In a short time frame solution, SL style measurement black boxes could be written to read PS data periodically (or on a supercycle basis), and inject the results into the existing software framework (tables similar to those used at LEP). In a longer range time frame, one could use the future publish/subscribe mechanism to support log-on-change as an option. In any case, the two groups should collaborate on any future upgrades.

6. Physics Applications

By far, the most complex software in a control system deals with the accelerator as an accelerator, and not just as a collection of I/O points. This type of software includes orbit analysis, emittance measurement, model driven beam steering, beta matching, chromaticity measurement and correction, and a long list of others. This type of software makes the most sophisticated use of the underlying control system, and can therefore become tightly coupled to a particular system.

There was insufficient time to do a great deal of analysis of this type of software, but a few general comments can be made. First of all, the LEP software is quite rich in functionality at the physics level. This is true of the Trim program mentioned already, but also includes a large package for beam correction (COCU — Closed Orbit Correction Utility) and many others. Most of this software is more coupled to the databases of LEP and their access library than they are to the front end server implementation, and deserve further analysis for re-use.

At PS, there is not as much software at this level, but there is one notable recent development, namely ABS — Automated Beam Steering. ABS was written intentionally to be loosely coupled to the control system. It has its own beam line description database, and the application itself is driven by this database. There are already some discussions underway to re-use this application for beam line steering at the Prevessin site. One suggested approach (endorsed here) is to have ABS use CDEV for device access, i.e. for the adaptor to the 2 control systems.

Pursuing this project would be a valuable testbed for sharing of high level physics software between 2 control systems.

7. Distributed Infrastructure

This category includes the software which other software may expect to have available in addition to the equipment access software. The major subsystems looked at include:

Timing System. The timing system is responsible for generating significant time events to control hardware and software processes. It includes software support for distributing events to client hosts with no timing hardware.

Alarm System. The alarm system is responsible for acquiring significant alarm events throughout the complex, recording them for later analysis, and presenting them to multiple operators. Alarms can be selected on the basis of source computer, alarm type, or other fields. For a given alarm, additional information may be made available to an operator, such as physical location of the alarming item, or help in resolving the problem (either a program or documentation).

Error System. The error system covers translating an error code returned by a front end or library into a readable string, logging errors to a file for later analysis, and presenting (a subset of) errors to one or more operators.

7.1 Timing System

The timing system spans front end hardware and software, plus network client software.

7.1.1 Hardware. As part of a joint development activity, SL and PS have adopted mostly identical hardware for their timing systems based upon older SL designs. (Each division continues to support different legacy timing hardware.) This timing system consists of a master timing system plus any number of timing receiver cards. The master timing node is implemented as a VME crate with a processor running LynxOS plus one or more master timing generator (MTG) cards. This VME crate is replicated so that there is a hot spare slaved to the master at all times, and special synchronization signals flow between these two. At each complex, there is one MTG card per major machine (CPS, PSB, LPI for PS, and SPS and LEP for SL). Each MTG feeds a cable connected to timing receiver cards Tg8 (or in the case of SL, older generation Tg3 cards). PS is currently using the new hardware, while SL has prototyped using it to replace the previous generation PC based cards.

7.1.2 Usage. The new MTG and Tg8 cards are able to support the differing PS and SL historical usages, with PS primarily employing a “telegram” style of usage, and SL employing an “event” style of usage.

To be more specific, the timing system may be used in both the traditional sense of sending time markers (events) to various pieces of equipment or CPU’s to initiate ramps, etc., and may also be used as a fast deterministic network carrying data from the MTG to various other systems. The telegram is a set of data events each carrying 16 bits of data tagged by the high 16 bits of the event. In the PS, these telegrams inform the remote equipment about what cycle is now being executed, plus what cycle will be executed next. In this way, the variations in the supercycle (as pieces of the spare supercycle are substituted for corresponding pieces of the normal supercycle) are conveyed to all equipment which needs this information.

As SL does not currently operate with a normal/spare supercycle, there is not as much decision making taking place on the fly in the CPU controlling the MTG predecessor, which is mainly used to transmit a previously generated pattern of events during each supercycle. Instead, SL uses the notion of accelerator modes (store/economy, pulsed, store, etc.) to conditionally execute parts of the event list.

Because of their differing usage patterns for the timing system, PS and SL have used (1) different firmware on the MTG (PS) and MTG predecessor (SL) and Tg8 cards, (2) different means (language and support software) for programming them, and (3) different means for handling the events generated on the distributed nodes (front ends and host client applications).

Each Tg8 card contains a 256 word content addressable memory (CAM) which is used to determine local interest in a particular event. The resulting data from the CAM is used to program 1 of 8 counters which delay the event locally, and may be used to interrupt the CPU. PS allocates

32 CAM words for each of the 8 output counters. SL has no such allocation policy, and any of the 256 words may be used to trigger any of the 8 counters. Also on the Tg8 cards, the PS has more code related to telegram processing, and SL has more software related to tracking the time of arrival of the events, and statistics about the events. A new version of the Tg8 driver from SL is due to be tested at PS soon, at which point a common firmware + driver will be in use.

On the MTG cards, the PS firmware is oriented toward multiplexing events and telegrams, whereas SL tends to process simple event lists (with conditional branching and looping). SL plans to take the PS developed MTG firmware for testing in the next run, although there is also some interest in further developing the firmware along the lines of the older system (modal and conditional event oriented).

7.1.3 Firmware and driver convergence. There are plans in place at this time to share both firmware and drivers, so convergence appears optimistic at this time. A strong commitment should be made to using a single software package for both PS and SL, with upgrades specified by a team from both divisions.

7.1.4 MTG Supporting Software. The software used to program the timing systems at the moment are quite different, with the view of the timing system (event or telegrams controlling cycle variations) dominating the appearance of the GUI and the database. In addition, PS modifies the programs of multiple MTG's in parallel. Software convergence here is not practical unless these views are first reconciled.

7.1.5 (Unix) Timing Client Software. Client software is of two types: software to manipulate the definition of the timing or analyse the performance of the timing system, and software to use the time events to do other work. For the first type of software, PS has a more elaborate interface in that each timing event definition is accessible as a first class device with properties, and the software is very much database driven. The SL software treats the timing system more as a software program (source code), and so manipulates the timing at that level. This software can only converge to the extent that the user interfaces used at PS are (made to be) acceptable to those in SL who need to modify the timing. It is unlikely that PS would be able to move in the direction of treating the MTG software as algorithmic code.

The other type of client software is that which uses the events. Both PS and SL have mechanisms for distributing timing system events/telegrams to clients running on computers anywhere on the network. This software consists of (1) a program which is essentially a direct client of the timing hardware/driver (e.g. using a Tg8 card), plus (2) networking software to distribute data via multicast and/or broadcast and/or direct socket connections to all interested machines on multiple networks, plus (3) a daemon on each node to receive the data and repeat it to all interested client applications on that node.

The application programming interface (API) varies, again, between one aimed at transporting events and one aimed at replicating a block of memory (the telegram in PS' case).

The SL software (SSM -- Software Synchronization Mechanism) views events as a 4-tuple consisting of (1) header (SPS or LEP), (2) event type, (3) cycle type (beam type), and (4) cycle number (within the supercycle). The configuration and administration functions are also somewhat customized to SPS/LEP operations.

The PS software (DTM -- Distributed Table Manager) transports named data tables, and supports 2 different API's, one for older programs, and one for newer programs. DTM itself has more capabilities for data transport than SSM, but event propagation does not need these features, and in fact these additional features are for the most part not used.

7.1.6 Recommendation. There is considerable duplication of software in the network infrastructure for SSM and DTM, and a combined system based upon a unified view of events (time markers plus small telegrams) with a common API would be not too difficult to achieve. The SSM package is cleaner in that it was designed to do only event distribution so it might be the better starting point. The other capabilities of DTM not used by a timing system (bulk data transport, sending data on change, etc.) should be moved to the standard RPC system and/or an asynchronous data monitoring system (discussed in “Subscription Capabilities” on page 5 and in “Moderate convergence” on page 7).

7.2 Alarm System

The *Alarm System* is that software which continually checks the validity of the state of the control system, and gathers alarm state information for logging and presentation to operators. Both SL and PS have alarm systems, albeit with different emphasis upon capabilities.

The SL alarm system, also known as the CERN Alarm System, is composed of the following major pieces:

1. software to test data and generate faults (functionality split between low level device code and a Standard Surveillance Program).
2. software to locally compress faults (multiple devices, oscillation) and detect if the upstream software has stopped working (corresponds to the Standard Surveillance Programs plus the Local Alarm Server)
3. a central server (redundant) to receive all faults and log them
4. multiple consoles, displaying a subset of all active faults (GUI)
5. a database of all possible alarms, providing a mapping between alarm number and a number of other items (priority, responsible person, location, action, display string, etc.)

This system emphasizes a central server and database, and employs a push event data transmission. This central database contains much information (such as location of devices) which is not available through the existing equipment access layer (for historical reasons); this information could be of value in other operator interface applications.

It must be noted that this CERN Alarm System is being used not just for the SL accelerator complex, but also for technical services, fire alarms, and other non-accelerator purposes. This demonstrates its robustness and adaptability, but also flags it as being difficult to evolve because of its current heavy usage.

The PS alarm system comprises:

1. standardization of how to obtain alarm state through the equipment access layer (using named properties of devices)
2. standardization of how to specify alarms (analogue limits, valid digital data patterns), again contained in the database, plus standardization of the mapping between error status bits and character strings per device class
3. operator GUI program, which polls a subset of the control system and presents alarms to the operator

This system emphasizes standardization, within the database and the equipment servers, of how to obtain alarm/status information, and employs a polling method to move data from the front ends to the displays. There is no central alarm server or logger at present.

7.2.1 Basic Convergence. As in the equipment access discussion above, there is a choice of the depth to which convergence is attempted. For a simple convergence, one could integrate PS alarms into the site-wide SL/CERN Alarm System, allowing a single view for LHC operations.

1. write a new client program which polls the PS complex (as the existing GUI does), and pushes fault states into an SL alarm server.
2. write an application which extracts information from the PS database, and formats it to be included into the SL/CERN Central Alarm database.

7.2.2 Advanced convergence. Simplify and then use the SL/CERN alarm system for PS. As a starting point, a more advanced equipment access layer is assumed (as in "Moderate convergence" on page 7).

1. Add a standardized mechanism for monitoring (subscribing to) alarms to both PS and SL controls, and create a Standard Surveillance Program which uses this subscription mechanism and then pushes alarm states upward. This new SSP could be used at both sites.
2. Modify the SL/CERN Alarm System to be able to use the more advanced features of the new, merged, equipment access. For such intelligent systems, there would no longer be a need to copy information from the device database into an alarm database -- the information would be obtained from the advanced equipment access instead of the private database as is now done.

7.3 Error System

When applications encounter errors, it is best if there is a standardized mechanism for logging those errors and/or presenting them to operators.

7.3.1 PS. PS has an error logging library, with a call similar to printf, including an argument to specify severity, and an ability to add a local error handler. An error log daemon receives errors from all applications, and forwards them to an error viewer application (one per console), which displays the program name, error severity, and error string; it also has some filtering capability.

The application frame (shell program used in console manager integration) also includes a convenience routine to pop up a window containing the error message prior to logging it.

7.3.2 SL. SL has a small library to allow daemons to print to a log file, automatically open a new file when a particular size is reached, and a signal handler to toggle a logging flag on and off. There is no support in this library for viewing errors from multiple applications in a single window.

7.3.3 Convergence. As these libraries address different needs, and are not tightly coupled to other software packages, they could both be easily shared.

7.4 Other Infrastructure

Another type of software infrastructure not really investigated is software for ensuring that clients and servers that should run all of the time are always alive. SL has a software manager (SM) which forks required applications and watches for dying processes and restarts them. PS has a command line utility with a similar capability for a single child process. Both of these represent sharable software.

8. Summary Recommendations

The recommendations below are in priority order, where high priority is given to tasks which have high impact on convergence or software sharing, have moderate impact but are easily accomplished, or are required by other tasks of high priority. The list below is a first cut, and will certainly evolve as others provide additional information. Each is marked with a proposed timescale for implementation where *short range* implies less than 6 months, *medium range* implies one year, and *long range* implies several years.

Overall, the flow of recommendations is to first develop a common equipment access library, second begin to share client applications (utility programs and libraries, and also physics applications), and finally, share in the development of next generation servers.

8.1 Prioritized List of Activities

1. (short range) Develop a common Java equipment access API. As Java is new to both systems, and holds high promise, and is being used on the AD project in PS and a GUILS upgrade in SL, it is ideally suited for collaborative activities at this time.

   Implementation should be at the moderate convergence level discussed above. It should hide the existence of the SL/Oracle settings database while merging capabilities in both the SL-Equip library and the SL operations library, and should similarly hide the PS/DSC equipment tables. Furthermore, it should support monitoring at the fine grain (device list, property) level. (See also the list in the API section.)

   (medium range) Work on merging the Java-CDEV and Java-Eqp packages to foster better world-wide collaboration, with anticipated benefits to CERN.
2. (short range) Set up a common source code repository for shared software, with clear procedures for adding and updating software. Initially, each package should have an owner, who is normally the only person changing the source code. CVS is recommended in that it provides good support for 2 distinct groups running on different development platforms.
3. (short range) Place the new MTG and Tg8 software into the source code repository, and place into production at both sites.
4. (short range) Port the SL Data Viewer, stand-alone or embedded version, to PS for use on the AD project.

   (long range) Develop an equivalent Java data viewer/editor package, most likely in the form of a java bean, which encapsulates vector editing, zoom and pan, plot and table view, and hot cursor tracking. Use a more java beans-like API, with event passing, instead of the existing library API.
5. (medium range) Timing event network distribution merger (SSM and DTM packages), with support for Java clients. Define events to optionally carry data so that PS telegrams can be carried in a way identical to other events. The resulting capability could be folded in as an extension to the equipment access API, first for Java, then for C/C++. Subscription should allow specifying which timing "system" to which one wants to subscribe.
6. (short range) For C/C++ shared software, implement the CDEV API on both PS and SL controls, with a view toward having a common equipment access API for C/C++. Propose extensions to CDEV where necessary. Choosing CDEV also allows re-using a

growing amount of software from other labs (stripChart, z-plot, correlation measurement, synoptic display, etc.).

7. (medium range) Implement a common subscription service, resident on each front end (LynxOS on powerPC only) OR on a Unix machine as a gateway. It should accept subscription requests at the (device, property) level, pushing data to the clients either periodically or on machine events.
8. (short range) Form a joint PS/SL team for beam steering, with the goal of having a common package for both complexes. Evaluate PS ABS at the SPS complex (beam line steering). Note: collaborative discussions have already started between members of the two divisions, and this activity is currently planned in an off-line sense, i.e. no direct connection to the control system.

   (medium to long range) Port the new ABS software to CDEV or other common API in order to make it an on-line tool for both PS and SL. Evaluate database structure for use at SPS and LEP. Either adopt the same table layout, or create an API to machine description information (as described above under API convergence), possibly extending the equipment access API or using CDEV message dispatch.

   This will be the first step in sharing optics software, and should be considered in that light. This project will serve to evaluate the types of machine description / discovery needed by these applications. Addressing these questions should be considered the major part of this activity.
9. (long range) Evaluate use of RESOLVE, from SLAC, for lattice diagnostics, and if it looks valuable, collaborate in turning that into an on-line application (their plans are to use CDEV).
10. (long range) Alarm system integration with a standard push client API, and a standard surveillance program using the new common API with either monitoring and/or polling. Adopt much of the server and network message format from SL Alarms, but with decreased dependence upon its private database for PS and SL controls, and instead an increased use of meta-data support contained in the new API. Alarm browsers should be integrated, including the live interaction capabilities of the PS browser, and having historical browsing capabilities.
11. (long term) Add data logging capabilities to PS, based initially on SL software, and using the proposed new Java/GUILS browser. Evaluate CDEV logging software plans for possible collaboration activities, especially at the browser level.
12. (long range) Evaluate the use of an object oriented front end, using CORBA as the network layer. Evaluate, as part of this project, existing OO servers within ECP/IT controls group, and at BNL.

9. Conclusions

There are no technical reasons which would prevent greatly increased re-use of software between the control systems for the PS complex, SPS, and LEP (and LHC). There is considerable duplication of effort represented in the existing systems, and there are areas which each system shines and where each could profit from the other’s capabilities.