RHIC/AP/53

ATR Commissioning Software Task Force Report

The Gang of 2 × Four
Ted D'Ottavio, Jörg Kewisch, Chris Saltmarsh, Smita Sathe
Todd Satogata, Don Shea, Steve Tepikian, Garry Trahern

December 16, 1994

Contents

1 Introduction 2

2 Design Methods 7

3 Physics and Engineering Conceptual Design 12

4 Control Flow and Machine Timing 21

5 Beam Threading 33

6 A Generic Device Description Scheme 45

7 User Interface Generalities 59

8 Partial Requirements List for ATR Commissioning 61

9 Summary and Recommendations 66

A Accelerator Configuration 71

B ADO Configuration Data Base 77

C Commercial SQL Front-End Tools 81

D Naming Convention for RHIC/ATR Devices and Signals 83

Chapter 1

Introduction

The Beam Injection Tests Software Task Force was charged with studying the software needed for the ATR tests, seen as a stepping stone or template for the larger scope of the full RHIC control system. This report outlines our avenues of exploration so far, presents the current analysis and implementation work in progress, and gives recommendations for the future on the ATR and longer time scales. This first section is an overview.

Why Do Analysis?

Top-down analysis techniques have been used in major software projects for perhaps 20 years, from functional decomposition to the current obsession with object-oriented analysis. Although such analysis is no guarantee of success, some points are fairly widely agreed upon:

- Postponing proper analysis increases costs and risks.
- Letting things grow from the low-level to high will almost certainly cause trouble.
- Effective analysis is not easy.

It is equally true that imposing a technique without regard to the cultural and technical realities of a given project can be just as much of a failure as no control at all. The control structure for a research accelerator does not lend itself kindly to a very strict top-down analysis:[1] the machine, during construction and commissioning, is itself an experiment to a

[1] This is amazingly well paralleled in the strain between the laissez-faire, run-time adaptable programming that "c" is designed for and the strongly typed, compile time constraints of OO purists. The glory of C++ is that it allows you the worst of both worlds.

certain extent: the requirements for the control system will change and parts of the system will be needed early as sub-systems come on line. Experience shows that control requirements change faster in the early stages of operation. The task force is trying to strike a balance: software must be built in response to perceived short and medium term needs, but longer term goals should be investigated both to prepare for higher-level software structures and to indicate course changes if lower-end components are seen to be inappropriate.

Here we mention a crucially important factor in a successful analysis: finding out what you want to do should be separated from how you are going to do it. For example, a good data design may be implemented in a flat file system, a filing cabinet or a relational database. That is an implementation decision. Allowing the data model of a particular database manager to drive the design is wrong. If it's been designed to be done in a particular way, that's the way it will be done and you risk ruling out more effective solutions before you start. "If your only tool is a hammer, all your problems look like nails".

What is the relevant scope?

There is a common habit in accelerator controls systems to build a very hard-defined line between 'the user' and 'the hardware'. This often works fairly well on such 'black-box' subsystems as vacuum or cryogenics (although even black boxes turn out to be not quite as uniformly black as first glance supposed) but becomes less evident when dealing with equipment interfaces that affect beam in more complex ways. This report assumes that the whole range of the software systems is the appropriate scope of analysis: thus it is appropriate that the accelerator physics group is strongly involved as their analysis also addresses the physics of the devices this software structure interfaces with. In terms of the *analysis*, the restrictions imposed by the reality of hardware and low-end software components should be minimized. That is not to say that there should be an implementation free-for-all. Implementation techniques, coding standards and robustness must differ at different levels of the system; these should be in response to engineering imperatives rather than arbitrary design constraints. A parallel exists in the data-hiding aspects of many top-down analysis techniques: it is certainly a good idea, in terms of the complexity and understandability of

a layered system, to hide the 'dirty details' from higher-level users. However, this becomes very dangerous without a complete understanding of end-use. If a low-end component hides information that is subsequently found to be important, either disruptive re-design and re-implementation must be undergone, or ad-hoc, pathological links put in. Or, often, the job is deemed impossible, or not important, or both. Because accelerator control systems will be asked to do things they were not designed to do, one must look carefully at any decision to shield information, however irrelevant it may appear to be during initial implementation.

Techniques

The members of the group have used a number of techniques in a number of areas: a brief summary is given here, with further discussion in the following text and in references. This is distinctly not an exhaustive list of analysis methods.

Extended Entity-Relationship (EER) modeling is for data design. A tool (**ErDraw**[3]) allows data modeling to be carried out graphically and subsequently implemented on a relational database such as Sybase.

Object Modeling[1] is close to the EER model, but includes actions on the modeled data, so that it is appropriate for implementation using OO techniques, in particular C++. A tool which contains OM[2] is in use.

Data flow diagrams[4] detail processes that act on identified data, where those data come from and where they go to. The OM tool mentioned above implements these graphical representations. We have used this technique for the highly abstract 'conceptual framework' for physics and engineering (Chapter 3).

Control flow. This is a more knotty area. State transition diagrams are often used to express when things happen, in what order. They are fine for fairly restricted systems (such as a Coke machine) but an accelerator control system soon becomes too complex for such techniques. We do have Glish[7], which is firstly a rule-based language for expressing, amongst other things, control flow. As it exists, and can implement control flow in a major part of the system domain, we are using it. It should be emphasised that, although the glish implementation domain does not cover the whole system, there is no obvious reason

why its descriptive power cannot be very widely used for design. Chapter 4 of this report is heavily influenced by Glish's way of expressing control flow. Perhaps not surprisingly; the original Glish, a distant relative of the current one, was designed exactly to deal with accelerator control flow, and lessons learned from its implementation influenced the design of the SPS/LEP control system. It should also be noted that the representation of control flow, other than textually, is a difficult problem which Glish does not address, although we are currently implementing a control flow simulator which may well help.

How seriously should these techniques be taken? The extremes are

1. Become fanatic, pick nits and split hairs or
2. Ignore it as irrelevant and a crutch for people who haven't the talent to do *Real Programming.*

The right answer is between these two, and a function of the people involved and the work to be done. That of course doesn't help a lot.

What is to follow

We give a brief summary of the remaining chapters in this report.

Chapter 2 describes the design methods that the task force has been using and would expect to use in the work that will follow.

Chapter 3 discusses the physics and engineering model that serves as a template for a data flow analysis of the accelerator.

Chapter 4 discusses the definition of machine events and introduces a way to model control flow. Since the data flow model of the previous chapter shows only static data transforms with no sequencing information, we must find a way to include temporal dependencies in our analysis. The work discussed in this chapter is preliminary in nature and needs to be addressed by all groups cooperatively.

The fifth chapter discusses the beam threading application within the context of the physics and engineering data flow model of chapter 3.

Chapter 6 describes a method to detail all the interconnections between devices in the accelerator. The control system needs to know how certain devices are "wired-up", and the

specification of such lists requires a general structure in which to do this. The "general device definition" tables provide this structure.

Chapter 7 discusses the requirements for graphical user interfaces. One principle of the design is that no GUI should drive the application; applications should be able to run whether a console operator interface is present or not.

The beam threading application was just one of many that the control system must be ready to perform next summer. Chapter 8 lists the requirements for the control system specified by Mike Harrison and Waldo MacKay.

Chapter 9 is the concluding chapter and includes itemized recommendations for implementing the analysis methods discussed in this report.

The appendices provide some detailed information about the data currently known that will be used by the control system. Appendix A discusses accelerator configuration data, especially including the *NameLookup* table, the generic cross reference for device names. Appendix B discusses the current status of the ADO configuration database. Appendix C describes the work being done to develop front end tools for database data entry from PC (and UNIX). Appendix D provides a definition of the naming convention for RHIC and ATR that is currently being used.

Final note: This document is a compilation of the efforts of eight people, and the text that follows illustrates the different styles of the authors. Consequently, there is some repetition of themes and perhaps some inconsistencies as well that the reader will encounter along the way. We hope the ride is not too bumpy.

Chapter 2

Design Methods

One of the goals of the ATR commissioning task force has been to develop a common language to describe both the functionality and data structure of applications appropriate for any level of the control system, from the front end computers to the operator consoles. Although we have not tried to be too strict about the use of such tools so far, we have settled on a specific set of ideas. We are using both the data flow and object diagram techniques from the methodology known as the Object Modeling Technique (OMT) of Rumbaugh et. al.[1]. We will define these two modeling techniques more explicitly below though one should read the relevant chapters of Rumbaugh's book for more detailed presentations of these ideas. In addition we have been using the Erdraw tool[3] provided by Victor Markowitz' group at LBL to facilitate entity-relationship modeling.

The use of a common set of descriptive languages that are fairly simple to learn allows designers, programmers and managers to participate in the evaluation and evolution of control system design. However, even if we have agreed on a common language there are potential problems with the use of certain words that are interpreted differently by different groups of people. For example, the word "Event" is used in different ways. In such cases we should agree on a common definition of these special words (so e.g., make "Event" the superset, and qualify with e.g., REL Event, MagTick Event, Glish Event...) or use different words. In addition, centralized documentation of the design effort using specific tools that implement methods such as OMT is also crucial. We have purchased a copy of the **Select OMT**[2] software tool for a PC and have been using it to develop models of the high level beam threading application. This is discussed in detail in Chapter 5.

Data Flow Diagrams

The three basic elements of data flow diagrams, *DFD's*, are processes, denoted by ovals and named by verbs, data stores, denoted by parallel lines and named by nouns, and finally data flows, denoted by arrows. Data flows are usually named as well unless it is understood that an entire data store is being transferred. Data flow diagrams are used to describe the functionality of a set of processes. Specific data elements flow out of a data store and are transformed by a named process which in turn passes data on to other processes or data stores. Data flow diagrams are not intended to model control flow, i.e., the temporal sequence in which the various transformations of data are taking place. These diagrams provide the necessary details for an understanding of what data and processes are needed. If drawn in sufficient detail, a data flow diagram should provide a clear explanation of what an application is supposed to do. A simple example of a data flow diagram is shown on the next page in Figure 2.1.

The structure of data flow diagrams is usually hierarchical; a specific process in one *DFD* can be exploded into many other processes and data flows at a lower level. This nesting of *DFD's* is necessary because most applications are sufficiently complicated that without the ability to hide the details of some of the component processes, one would not be able to understand the application as a whole.

Object Diagrams

Object diagrams, *OD's*, in the Rumbaugh methodology provide models for both the data stores and the processes used in the data flow diagrams. They are designed with the aim of implementation in object-oriented languages such as C++. A given object in an *OD* contains both the list of attributes (and their data types) as well as methods that are associated with that object. The choice of which methods to include in the object should be clear from the data flow analysis. Relationships (association, inheritance, etc.) between objects are modeled by connecting directed lines between objects. The usual 1–1 and 1–Many relationships familiar from Entity-Relationship modeling techniques are supported by the object diagram technique.

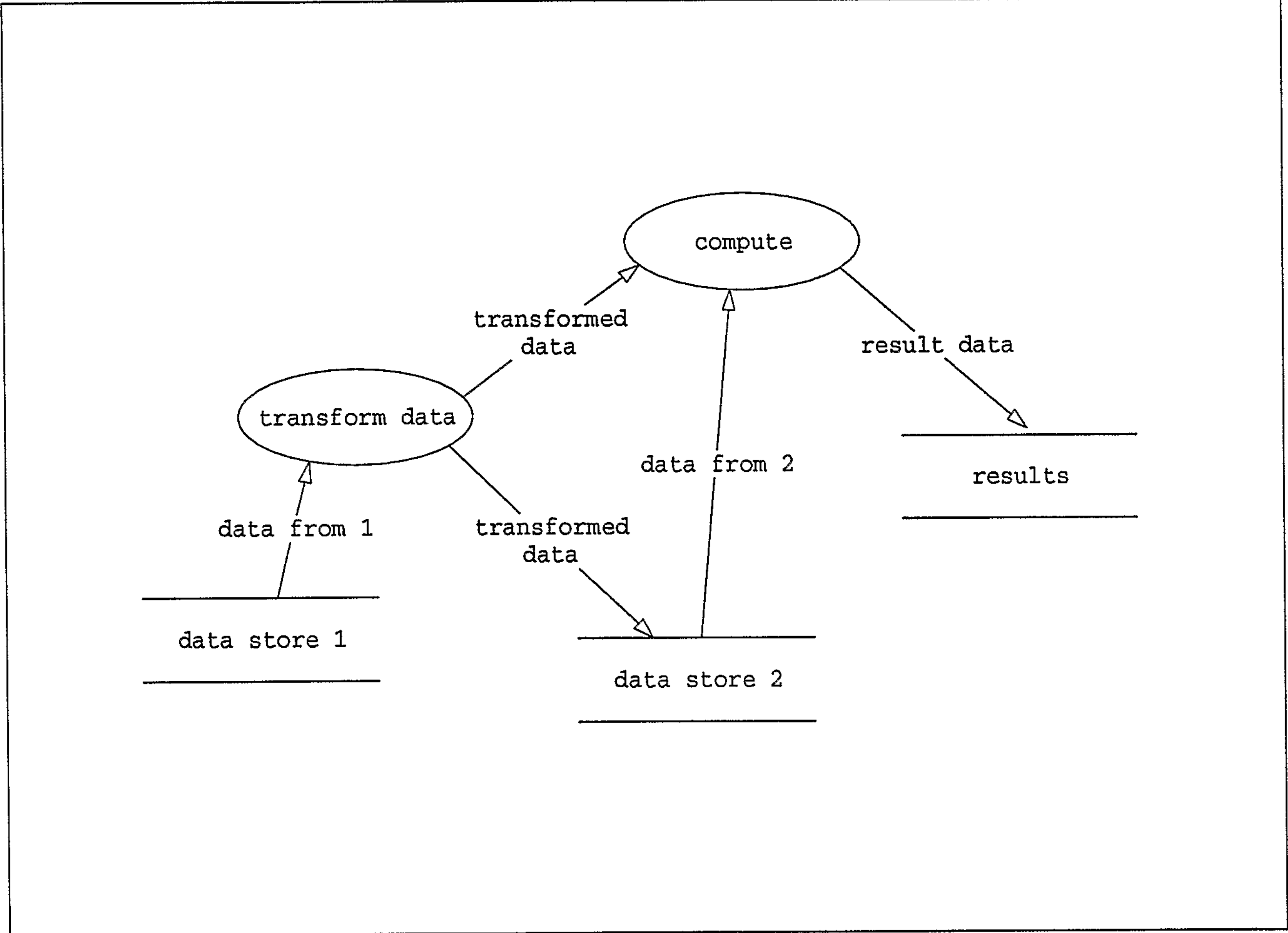

Figure 2.1: Generic Data Flow Diagram

Given a complete specification of an object diagram, one can generate C++ code for the classes(objects) as well as the SQL code needed for database table creation via the code generation tools provided with the **Select OMT** toolkit.

Data Dictionary

The data dictionary is principally composed of the elements of Object diagrams. These form the basic data structures that are transformed by the processes listed in the data flow diagrams. Using the case tool it is possible to associate objects in the *OD* and the *DFD* via the data dictionary. A clean analysis of the application should in principle lead one to define objects which are properly isolated in the *DFD*.

Control Flow

Control flow, timing and sequencing are discussed in the Introduction as well as in Chapter 4. The *glish* interpreter provides a succinct language for simulating and/or implementing control flow. We will be using *glish* to construct models of control flow.

Entity Relationship Modeling

Markowitz has implemented a methodology known as the Extended Entity Relationship (EER) model in a tool called **erdraw**[3]. We have been using this tool to design databases prior to the existence of the ATR task force. (See section II of Chapter 5 for a more extensive discussion of EER modeling.) **erdraw** provides a visual interface to define data objects and their relationships to other objects. It does not support object methods as in the OMT methodology (nor any of the other design techniques such as *DFD*, etc.). However, **erdraw** does generate SQL code for SYBASE including all the referential integrity constraints in the ERR model. Finally, the last great advantage of **erdraw** is that it is freely available from LBL.

Since the **Select OMT** tool also provides ER modeling in the version we have purchased, we will be evaluating whether we can work completely within the OMT methodology or not. The SQL generated by **Select OMT**, while ANSI compatible, is not tailored for SYBASE

nor does it generate the code needed to enforce referential integrity, a nice feature of **erdraw.**

Chapter 3

Physics and Engineering Conceptual Design

I. Motivation

The motivation to develop a new system for RHIC is the desire to provide a system which goes beyond the functionality of available systems and provides support for the operator in understanding and improving the machine.

Traditionally control systems have a well separated data acquisition level and application level. The data acquisition level provides basic read and write functions, data conversion, alarms (limit checking) and networking. A general display and editing facility (Parameter page) is also included. In some cases accelerators have been commissioned just with these tools. However, the lack of better tools promotes twiddling of the machine without understanding.

The application level provides the necessary operational support that makes the commissioning less painful. Application programs use knowledge of the accelerator to automate complicated operating sequences. Examples are beam threading, orbit correction, tune correction and chromaticity correction.

Often the application level is designed and written as a collection of independent programs with little or no interaction. They are merely tools for better twiddling. In a well-designed control system application programs are embedded in an environment that promotes the understanding of the machine.

Operation in such an environment follows this diagram.

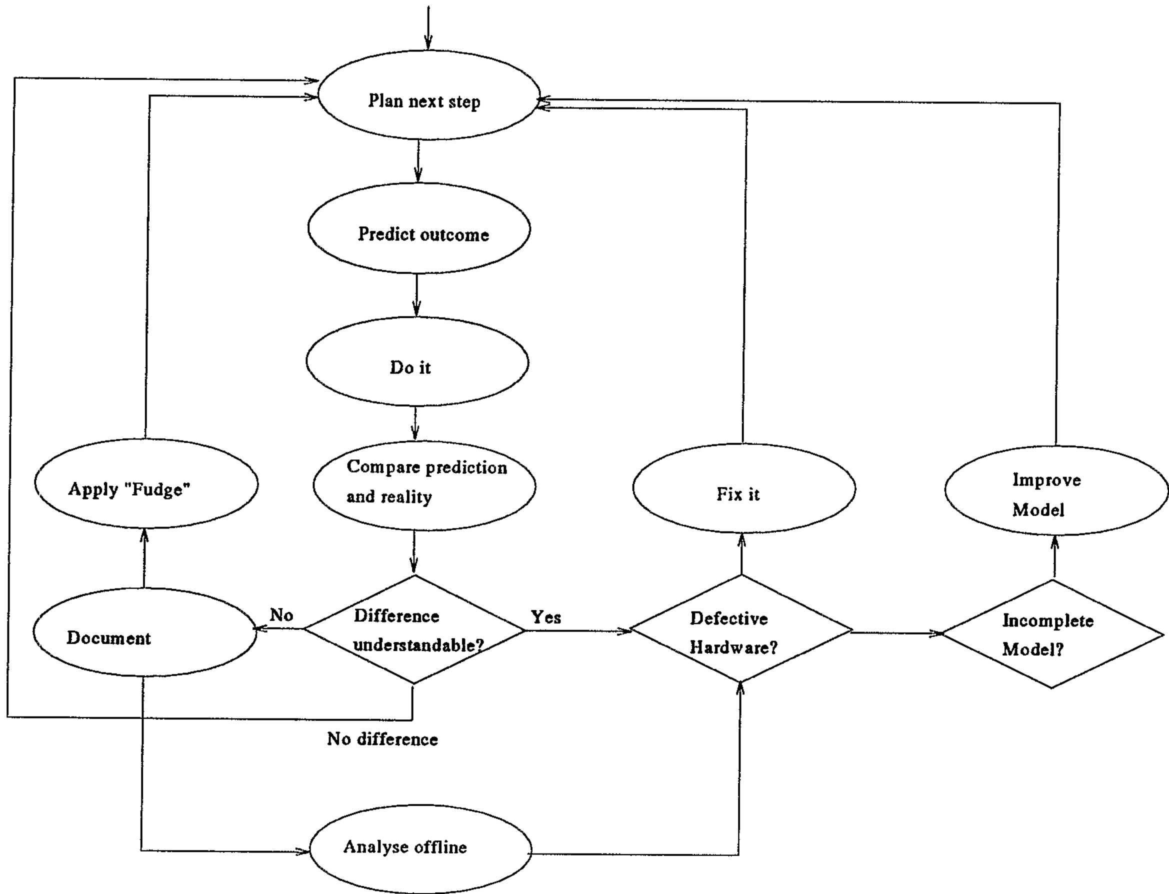

Figure 3.1: Preferred Operational Environment

Instead of twiddling until something works, the operator first plans what he wants to achieve in the next step. The commands are given on a high level. For example, the operator sets the tune or the orbit of the machine to the desired value and not the magnet currents. The control system then calculates the dependent parameters like magnet strength and current and predicts the effect of such a change on all observables of the machine. This calculation is based on a model of the accelerator which includes all knowledge accumulated so far.

If the change is consistent with the limits of all parameters (i.e. the magnet current can be delivered by the power supplies, the beam position stays inside the beam pipe, the tune does not cross an integer resonance...), the operator commits the change.

The control system implements the new settings and measures the response of the machine. If the measurements differ from the prediction, the control system aids the operator in understanding the machine behavior. If the reason for the difference is understood, it is

taken into account by repairing the machine or updating the model.

If the reason for the difference cannot be quantitatively assessed, the machine is corrected by applying "trims", changes in the machine settings not included in the model. These changes force the measurable behavior of the machine to agree with the model. For example, orbit errors caused by misalignment of quadrupoles are corrected by powering correction dipoles whose design strength is zero. The control system aids the understanding of the machine by separating trims from settings and documenting the machine operation for later off-line analysis.

Experience has shown that a large amount of the useful application level programs are written during the commissioning phase of the accelerator when experience with operating the machine is translated into operation procedures. At such times there is great pressure to produce quick and dirty solutions. A coherent system can only be produced if an environment is in place that supports the described operating philosophy.

II. Definition of terms

For each accelerator parameter we define a data object which contains the following data:

Parameters are the set points that describe the desired state of the accelerator. Parameters are a function of time. The parameter value at any time is obtained by specifying the values at step-stones and using the appropriate interpolation method. In most cases the step-stones will coincide with the events on the accelerator event line. The control system can modify future step-stones, while present and past step-stones are not changeable.

All quantities that describe the state of the accelerator are parameters. The parameters are therefore redundant. A system of processes ensures that the parameters are consistent (this is discussed later.) This allows the operator to describe his goal directly. He can set any parameter and leave the task of finding the corresponding hardware settings to the control system.

Measurements are the actual readings from the machine hardware. (For most parameters a direct measurement is not available.) They are conceptually independent from the parameters with the same name. Although some hardware devices include the setting and measurement of an accelerator parameter in one module, this is the exception.

Trims are changes to the accelerator parameters that enforce desired behavior. Trims

are only used in objects where a measurement is available. The goal of the operation is to make the parameter and the measurement the same.

Method data are data that determine how to convert dependent parameters. Methods often change during operations. For example, given the desired orbit the method data describes which correction dipoles are used to move the orbit.

Acceleratorknowledge is configuration information about the accelerator. This data changes only when the machine or the control system is reconfigured. Examples are lattice information, host names, power supply names and magnet data.

III. Data flow diagram

The goal is an environment where new parameters of the accelerator can be easily included in the control system. Below we define a "Parameter object" which handles a single accelerator parameter and uses the above items. In this object stubs are provided to define the following processes:

1. calculation of all dependent parameters, if the new parameter is changed.
2. update of the new parameter if a parameter on which the new parameter depends changes.
3. measurement (and generation of an alarm) of the parameter, if possible.
4. prediction of trims.

For the description of the control environment we will use data flow diagrams. (See Chapter 2.) The control flow is not presented in these diagrams. At this level it is assumed that processes are infinitely fast so that sequencing becomes unimportant. The sequencing will be investigated in the implementation phase.

The diagrams will not contain the user interface. It is assumed that all data in data stores is available for display. It is also assumed that all commands to the system can come either from a graphical user interface or from a sequencing mechanism. We plan to use *glish* as such a sequencer.

IV. Parameter conversion using the data objects

For simplicity we first describe an object that includes no measurements and therefore no trims:

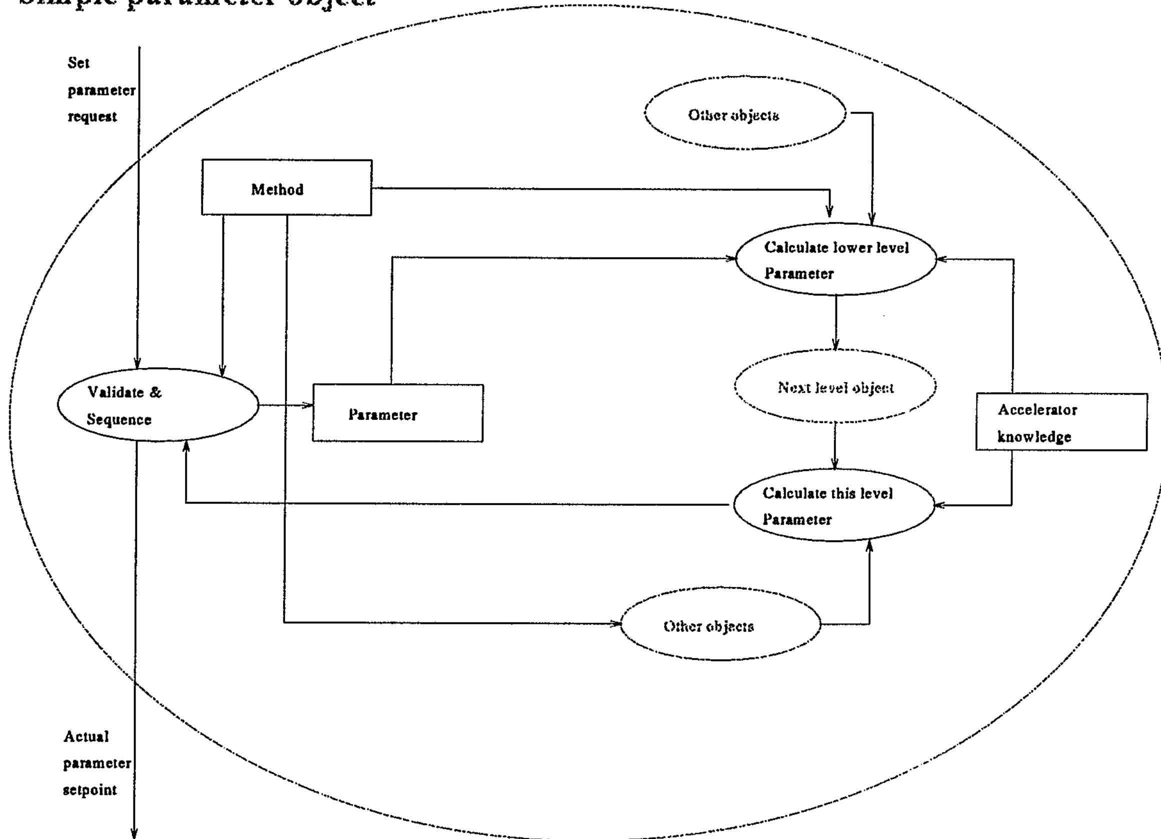

Figure 3.2: Simplified Parameter Object

A set-request from the operator or sequencer or a higher level object is first checked for semantics and for range in the "validate and sequence" process. Method data is used for this check.

If the request is valid, the parameter is set in the parameter data store. The "validate and sequence" process administers the step-stones in the parameter store. The parameter is passed on to the "calculate lower parameter" process. Method data is used to determine which lower level parameters are used to do the change. The process may need also other parameters and accelerator knowledge for the calculation.

The calculated parameters are sent to appropriate object(s) which internally have the same structure as this object. The design is recursive. The lower level object returns the set

parameter, or in the case of error, an error message and the best achievable parameter value which is passed to the validate and sequence process.

The validate and sequence process checks the success of the set operation. It updates the parameter value in the data store and returns the value or best value to the next higher level.

The "model" of the accelerator resides in the "calculate lower level parameter" and "calculate this level parameter" processes. It is important that these modeling processes occur on each level and are by design independent from each other. This allows implementing new parameters and extensions of the model in a flexible way. In implementation the same program or process might be used for the different levels. An optics calculation process might be set up as an "persistent" client and serve different parameter objects.

The accelerator knowledge data store provides the basic information for the model. It contains lattice information, magnet data, etc. Although this data changes not very often, it is a part of the data flow.

The recursive design suggests that parameters can be organized as a tree, where each object is a node with all internally used objects as leaves. However, it turns out that there is no natural structure for this tree. Whatever parameter is set by the operator or sequencer is the top level. The tree structure must therefore be dynamically configured for each operator or sequencer command. This configuration is contained in the method data stores. The device definition database can be used as a tool to define these connections. For more on the device definition database, see Chapter 6.

V. Treatment of measurements

The system described so far allows the generation of a consistent set of redundant parameters. If the model used is correct, the parameters give a complete description of the machine. Unfortunately, real life is often disturbingly different than what it ought to be, and the measurements of parameters differ from their desired values. The parameter object therefore contains measure and trim processes. (See Figure 3.3 at the end of the chapter.)

In addition to the parameter store the parameter object contains a trim store. The goal of operations is to predict the best trim, so that the machine behaves as described by the parameter. The trim is predicted on the basis of past experience. The "Predict Trim" process calculates the trim for the future step-stone from the measurement of the past or present and

the parameters and trims that lead to those measurements. The "combine" process combines the parameter value and the trim value into a "parameter to be installed" value. In most cases the combination means just adding the two values, but more complicated combinations are possible. Instead of the desired(requested) parameter, this "parameter to be installed" is passed to the "calculate lower parameter" process. The returned values from the dependent parameter objects need to be "un-combined" before they are sent to the validate and sequence process for interpretation.

An example is the orbit correction procedure. The procedure starts with a request for a centered beam (zero beam position) in the next step-stone. Using the latest measurement of the orbit, the best trim is predicted to be the negative value of the measurement (i.e. a linear machine is assumed.) The "combine" function adds the trim to the desired orbit. The desired orbit can be nonzero for injection bumps and background minimization. The "orbit to be installed" is therefore the negative value of the measured orbit. The "calculate lower parameter" process calculates the strength of the corrector dipoles using traditional orbit correction techniques. Which method and correctors are used is defined in the method data store. The Twiss functions of the machine are also installed as a parameter object and the correction process can use their values.

The calculated corrector strengths are sent to magnet strength objects. If the required strengths cannot be produced by the magnet power supplies, a magnet strength object returns the maximum strength possible and an error. The "un-combine" process subtracts the trim values and returns the expected orbit after correction to the validate and sequence process.

The validate and sequence process reviews the results. It may decide a value is close enough or totally out of whack and pass this information on to the calling process, or it may modify the method data for the "calculate lower level parameter" process and try again. The method data for the "validate and sequence" process determine this action.

VI. Initial List of parameters

An initial list of parameter objects used for the ATR.

Parameter	Parameter changed to set this parameter	Other parameter used for calculation of lower parameter	Other parameter used to calculate this parameter from lower parameter	Directly measurable
beam position	correction dipole strength	Twiss parameter	magnet strength	yes
beam size	quadrupole strength	Twiss parameter	magnet strength	yes
Twiss parameter	quadrupole strength	Twiss parameter	magnet strength	no
Dispersion	quadrupole strength	Twiss parameter	magnet strength	yes
Energy	Dipole strength			no
Dipole Strength	Dipole field	Energy	Energy	yes
Quadrupole strength	Quadrupole field gradient	Energy	Energy	no
Correction dipole strength	Dipole field	Energy	Energy	no
Dipole field	Magnet current			no
Quadrupole field gradient	Magnet current			yes
Magnet current	Power supply current			yes
Power supply current	WFG tables	Step-stone time intervals	Step-stone time intervals	yes
WFG tables	Power supply current	Step-stone time intervals	Step-stone time intervals	yes
Step-stone time intervals	Magnet current gradient, magnet current 2nd order gradient	Power supply current	Power supply current	yes
Magnet current gradient	Step-stone time intervals	Power supply current	Power supply current	yes
Magnet current 2nd order gradient	Step-stone time intervals	Power supply current	Power supply current	yes

Table 3.1: Sample parameter object list for ATR

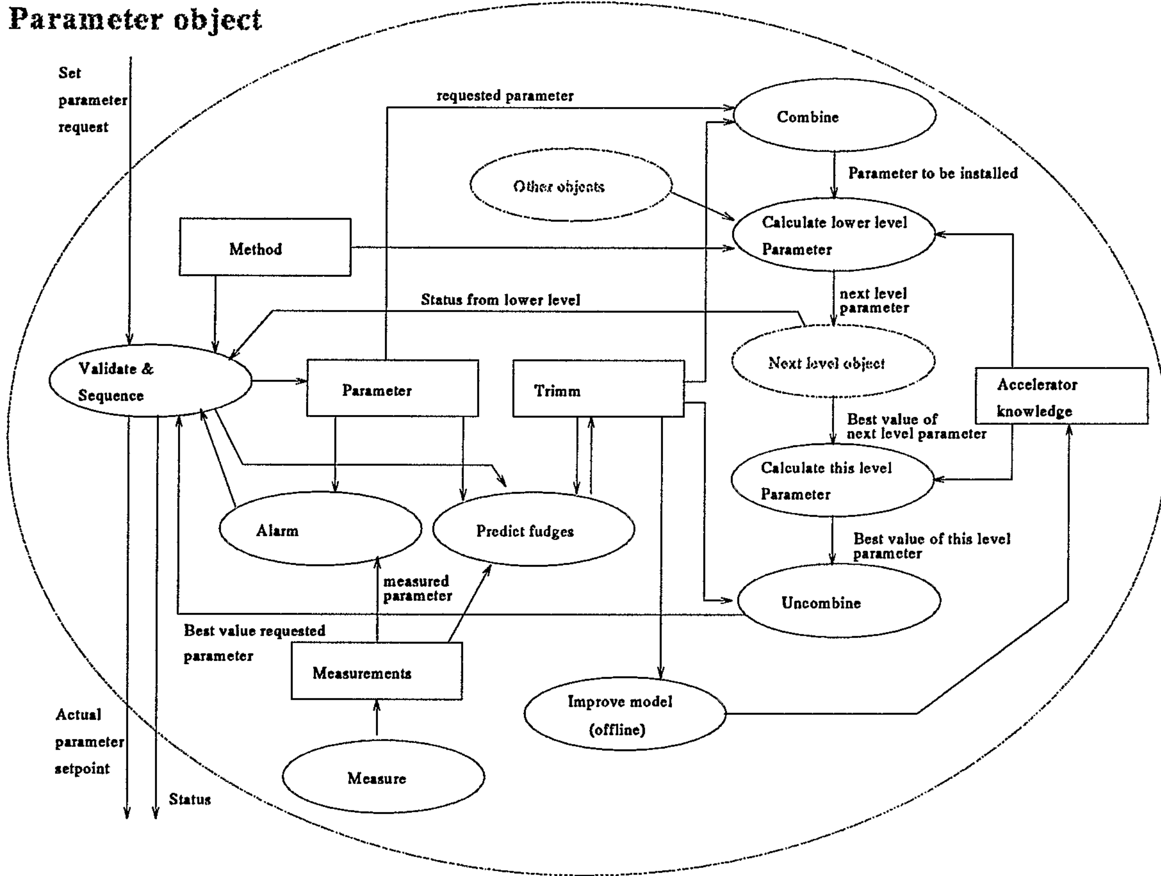

Figure 3.3: Parameter Object

Chapter 4

Control Flow and Machine Timing

Within accelerator control systems control flow at the "high" or "application" level does not often appear to be explicitly considered - at least, not in the early days, although formal systems are often introduced later in an accelerator's life. Of course control flow is around, in the procedural flow within programs, in the actions of operations crew and, in theory, in the procedures they are supposed to follow. Practically every accelerator does, however, have a hard-nosed control flow system, called the event train, or timeline or some such expression: hardware triggers broadcast over the machine(s) from a small number of sources and co-ordinating the actions of many component devices. Time scales are typically at the microsecond level for beam-related operations and at the millisecond level for the magnetic machine. Arranging that these triggers go out at the correct times and cause appropriate actions on reception tends to be complex and often hidden from the "high" level control: where high level *must* control things, such as in the count-down sequence for a collider fill, some pretty hairy code tends to get written.

Further control flow exists at the low-end device level, in hardware timing chains and in the operation of realtime processes; again very often hidden from on high. Such hiding of control flow may be appropriate and certainly layering is needed by rational implementation schemes: central realtime event distribution is not possible directly from current workstation and network implementations. We believe that this level of control flow *should* be understood, and where it is buried out of the understanding and/or control of high level operation it should be done as a decision based on a complete design.

In our case we have an opportunity to integrate tools to aid such problems into the control

system design. The first step is to break away from implementation constraints and see if the operation of the accelerator, in terms of its control flow, can be described in terms of well-understood events and the processes they communicate with. This is potentially a big deal: most texts on formal analysis will recommend some kind of state analysis when you get to the control flow part of the problem. While this may be fine for a Coke machine, and even for a nuclear power station if you have the time and effort to do a state analysis, we do not believe it is relevant for an accelerator. That's no reason to ignore flow control and hope it will go away: it won't: it may become a serious problem instead.

[Of course, we believe that such an analysis can and should be done, and we point to existing colliders and the sequencing that has been applied to them to prove it.]

As it happens, we do have a language in which we can try to express such control flow; it uses the notion of "events" and has familiar application in accelerators, although the domain of application is being expanded. It is rule-based so we do an end-run around the problems of a state analysis while admitting some of the interesting uncertainty of research accelerator control. (And, of course, we have code that will actually implement this expression of control flow, so that an analysis will go to implementation almost directly, at least in processes running at the workstation level. This does <u>not</u> mean that all control flow must be mediated by Glish.)

[An aside. A graphical mechanism for describing control-flow, called Petri net theory, exists and is possibly useful in this context. I have looked at it a bit, and it addresses the sort of problem Glish is designed to deal with. It does not *implement* sequencing however: it is a theoretical system to describe event flow, with a graphical notation for a better understanding of what is happening, and a fairly large body of theoretical work which enables one to identify, for example, state machines and potential live- or dead-lock in a given system. The other place to look for tools to deal with this is undoubtably in network design and maintenance. Telephone companies do it.]

To start, then, we look at what an "event" *is*. After that, we can see if the control constructs of Glish are up to the job a describing accelerator operation.

Events. What is an event?

Glish has a direct answer - an event is a name/value pair, where the event name is an ASCII string and the value is an arbitrarily structured, but named and typed, data record.

These events can be sent from process to process. The connection is at root *asynchronous*: a receiving process in general will not know exactly when a given event may arrive, and indeed may know very little a priori about the identity or value of an arriving event. For the accelerator, we have to change this definition - hardware events are typically *not* named with an ASCII string and have very restricted value sets, but the principle is the same. So we start off by abstracting, and presume that particular sorts of events will be seen as specializations of the abstraction.

Given this, the definition of event can be: An identified piece of data which comes into existence at a specific time and may be communicated to event consumers.

And its environment? It will connect to event source(s). Probably not necessarily all known a priori, but an event cannot be identified without at least one source being identified.

It will probably go to event consumers, but can be identified before any of its consumers.

By a "source" we do not mean a physical source (such-and-such a module, or program), but the "conceptual" source... "such-and-such" happening: a calculation finishing, an error condition arising (or perhaps being detected), an operator intervention happening, a kicker firing.

Now a look at the *types* of events we see in an accelerator. From this we may find out how to characterize the general event, and from this the specific characteristics of the specializations:

Event types are :

1. - Machine ticks: synchrotron tick, turn, injection, luminosity delta...

2. - State change : Injection warning, hand-over to ops, start low-beta squeeze, current out-of-tolerance...

3. - Time ticks : millisecond, hour...

4. - Information : B-field, intensity, orbit...

[Is an alarm a distinct type of event? No, we think that it is a *use* to which an event is put.]

The above indicates we have three main entities to define:

1. The events, which have types as above and some properties such as rep rate, value structure, etc.

2. the event sources, and

3. the event consumers.

From each event we will need to point to the properties that are common to all event types, and to those properties that categorize the separate types. For many, if not all, we will also be able to point fairly directly to the event source(s) and to some at least of its consumers. This will show us the implementation already foreseen for many events: it may also indicate possible difficulties due to these implementations.

Note that an event *may* be transmitted on several domains. Thus "Start low beta squeeze" may be defined as "0xFC" on the Rhic Event Line (REL), and as "LowBetaSqueeze, True" on a glish socket: these are the *same* event, just the delivery mechanism is different.

In what follows, we are going to argue by example. The actual examples will have some sense in them but they won't be *right*; building the complete event definition structure is complicated and will need thought and a lot of research into what is planned. What I'm after here is describing a structure within which that work can be done. Note also that this structure should not be dependent on transport mechanisms for these events. That can be found out afterwards. We do know that we have REL, RTDL, BeamSync, glish messages and so on but here we're concerned with the *content* of events and their interrelations, not how they get delivered.

Start with Event, whose main elements are

- Name: such as Second, CycleSecond, WarningInjection, Abort, BField.

- Type: one of

  TimeTick [CycleSecond]

  MachineTick [TurnTick]

  External [Abort]

  Info [BField]

  StateChange [StartCycle,LowBetaSqueeze]

- Derivation : one of

  Primary

  Derived

- A Value :

  "Second" has no value

  "CycleSecond" has value as an integer number of seconds from event "StartCycle"

  "UnixSecond" has an integer number of seconds from Jan 1, 1970

  "BField" has a 16-bit number representing the programmed main dipole field

  "LowBetaSqueeze" has True or False (start and stop)

  ....

- One or more Sources, of which more later. (Note that for an event to be "synchronous" or "asynchronous" is NOT an intrinsic property of the event, but rather of its source. Thus "LowBetaSqueeze, True" may come from a pre-programmed table loaded into the master event generator, or from the asynchronous push of a console button - it depends on circumstances. The former should be used in production operation, the latter when commissioning, tuning etc.)

- Zero or more Consumers, of which more later. Consumers are likely to be much more dynamic than other elements. For instance, when doing injection tests the event "LowBetaSqueeze" will have no consumers - knowing this is a good thing because one then knows that one can fool with that event's source(s) and anything else that is dependent upon the event.

- A Definition.

  Just for now, we deal with the definitions as commentary; but we'll find that definitions will start to group themselves. Many machine events, for instance, will be inter-dependent (hence the "Primary" and "Derived" flags) and their definitions will come, ultimately, from the lattice calculations. Some events - especially machine synchronization - will be defined in terms of how actual signals are treated by real hardware.

Concentrating on the definition of events is supposed to raise, and answer, real questions of where they come from, where they go to - so, for example, one can recognize if and when simulated events will be needed, and what their characteristics should be.

- One or more implementations - REL, RTDL, VME Interrupt, Glish...

OK, so let's start trying to describe some real events - remember that I'm just trying to get a structure in mind, so reality may already have made some of the following untrue...

Name	:StartCycle
Type	:External
	:Primary
Value	:none
Sources	:AGS, simulation
Consumers	:legion, including time counters, initialisations, ramp play outs...
Definition	:Start of RHIC magnetic cycle

That makes us think of something...

Name	:WarningStartCycle
Type	:External
	:Derived
Value	:none
Sources	:AGS, simulation
Consumers	:Things that have to be ready before the magnetic cycle starts – function generator resets, perhaps...
Definition	:100 msec before StartCycle

...and the first argument: is WarningStartCycle primary, and StartCycle derived?

Name	:HoldState

Type	:StateChange
	:Primary
Value	:True/false
Sources	:
Consumers	:
Definition	:A event to trigger an energy hold state, in which magnet and rf ramps hold the machine steady. (Some things may still have to work actively...)

Name	:ElapsedMagTicks
Type	:TimeTick
	:Derived
Value	:Integer, number of active 720Hz ticks after StartCycle
Sources	:
Consumers	:
Definition	:A count of the number of synchronous 720Hz ticks within the cycle, zeroed at StartCycle.

...but see...

Name	:CycleMagTicks
Type	:TimeTick
	:Derived
Value	:Integer, number of active 720Hz ticks after StartCycle
Sources	:
Consumers	:
Definition	:A count of the number of synchronous 720Hz ticks within the cycle, zeroed at StartCycle and NOT counting ticks within cycle hold states.

Name	:WarningInjection
Type	:External
	:Primary
Value	:None
Sources	:AGS, simulation
Consumers	:
Definition	:70 μsec before any injection

Name	:WarningInjectionBlue
Type	:External
	:Primary
Value	:None
Sources	:AGS, simulation
Consumers	:
Definition	:70 μsec before any injection destined for the X-line (and beyond)

Name	:WarningLastInjection
Type	:External
	:Primary
Value	:None
Sources	:AGS, simulation
Consumers	:
Definition	:70 μsec before the last injection

Name	:StartAcceleration
Type	:StateChange
	:Derived
Value	:none
Sources	:Programmed

Consumers :

Definition :100 CycleMagTicks after WarningLastInjection

The context.

To try to find a context for the above discussion, we offer a story of early RHIC operation. It is of course inaccurate, but serves to illustrate the ways in which a well-structured and well-understood event system can tie together many levels of the machine.

The machine is running physics. Filling takes 2 to 4 hours, and luminosity lifetime and background make a fill pretty useless after 4 to 8 hours.

1. Automated filling is not running; operations sees occasional bad injections which, if left in, cause unacceptable background. The bad shots are rare enough that the cause is being investigated while attempting physics running. Accordingly, operations requires magnet current measurements for all the injection line and the ring injection elements, together with injector beam parameters, beam trajectories, turn by-turn measurements and immediate orbit for each injection. All data are stored for later attempts at correlation, and beam information is fed online to displays and a beam quality program which flags bad injections. The machine is held at injection energy, and only when the crew chief is satisfied will the event train be let out of its injection loop and move to the acceleration and storage sequence.

2, Acceleration, beta squeeze and hand-over to physics is automated, except that a possible hold-point may occur just after the beginning of acceleration as a new front-porch ramp is being used, and indications of beam loss will cause an automatic hold. This may be released by operations if the beam settles down. In any case, all main power supplies and low beta supplies must have fast (720Hz) monitoring for the 2 seconds around the start of ramp. Beam quality is measured by a central program using the standard beam acquisitions.

3. During the beta squeeze, quadrupole current measurements are taken together with luminosity and background counts. Together with fill data on luminosity lifetime and background rates, these data will be used to investigate fill-to-fill variations and to provide data that can be used to judge appropriate fill times to maximize integrated luminosity. This is offline data; no-one is around to care for it, and operations is too busy; it must be completely automated.

Here's the pseudo-code to describe the first sequence. It looks like Glish. That doesn't mean to say it will be implemented in Glish; some of it happens at the FEC level.

```
client("ATRPowerSupplyADO", "AllYLine") // Tell the FEC's of the data
client("ATRBeamPosADO")                 // that will be required
client("ATRBeamLossADO")                //
client("OpsEvents")                     // Sent out by operations crew
                                        // from programs or button-pushes
client("MachineEvents")                 // Most of these will be
                                        // transmitted on the REL: they
                                        // will also be required at console
                                        // level.
client("InjectionAnalysis")             // Console program
client("InjectionDisplay")              // Console program
client("InjectionArchive")              // Service program
client("AlarmService")                  // Service program
client("AGS")                           // For informing our injector
                                        // what's happening

whenever MachineEvent.InjectionWarning do
  {
  ATRPowerSupplyADO.GrabData()
  ATRBeamPosADO.GrabData()
  ATRBeamLossADO.GrabData()
  }

whenever ATRPowerSupplyADO.GotData do // Just to archive
  InjectionArchive->NewData(RunNumber,InjectionNumber,$value)

whenever ATRBeamPosADO.GotData,ATRBeamLossADO do // to archive and online
```

```
  {
  InjectionArchive->NewData(RunNumber,InjectionNumber,$value)
  InjectionAnalysis->NewData(RunNumber,InjectionNumber,$value)
  InjectionDisplay->NewData(RunNumber,InjectionNumber,$value)
  }

whenever InjectionAnalysis->BadInjection do // tell everyone
  {
  InjectionArchive->FlagBad(RunNumber,InjectionNumber)
  AlarmService->Alarm("Bad Injecton",RunNumber,InjectionNumber)
  }

whenever PSSurveillance.Warning do // promote warnings to alarms for now
  AlarmService->Alarm($value)

whenever MachineEvent.BeamInTransfer() do
  {
  InjectionAnalyis->RealBeam()
  }

whenever OpsEvent.RejectInjection
  {
  AGS->Hold()
  RunNumber++
  InjectionNumber = 0
  }
```

...and so forth. This isn't a prescription of what has to happen; it is intended as an example and a template. By looking at realistic scenes like this, we can identify real questions (when and how do we get injection warning events? What's a RunNumber? Is it needed?) and analyse control flow in a way that will be accessible to all sides of the system. In

particular, such exercises *now* can help define the base componenets of the system - those that must be implemented at the FEC level - so that the required functionality exists when a new event, dependant upon the base set, must be defined and used by operations.

We could work on the other sub-sequences in our story, but this would become tiresome. At this point we ask – Does this express a valid concern? and if so is this a mechanism to address it.[1]

[1] Adding to the earlier note on page 2: the stress in accelerator control systems between runtime flexibility and 'compiled-in' security is just the sort of problem that exercises the programming language mavens. The subject is quite complex, and it is worth studying as it has fairly direct applicability to our problems.

Chapter 5

Beam Threading

I. The Generic Parameter DFD for Accelerator Controls

A fundamental presumption in this paper is that the RHIC control system should be as modular as possible, with well-defined mechanisms in place for combining various processes in new applications, which can then themselves be used by other applications. Central to this philosophy is a model in which each process in the system (such as high-level applications and ADOs) can, at least in principle, communicate with any other other process. A possible implementation, though not the only one, is a network of Glish clients.

In this model hierarchies and directed graphs of control objects naturally form, with simpler objects being grouped in successively more complicated ways and high-level objects intertwined in mutual dependence. There are several different functionalities which must be embedded at the low-level; however, there is no reason to presume that their interface should be functionally any different from the interfaces in high-level controls. Simplicity and consistency of interfaces are equally as important as flexibility and power; reading measured and predicted beam positions are the same sorts of requests even though the implementations are quite different.

Figure 3.3 in Chapter 3 shows a DFD for a generic parameter process as developed by the task force. This represents one process in a process graph, ranging from simple low-level processes such as control programs just above the hardware driver level up to high-level control points, or parameters, of an accelerator, such as the orbit and chromaticities. This diagram can in principle encapsulate *any* activity within the system, from internals of drivers

to an orbit correction algorithm to a simulation of beam image on flags.

Because a process communicates with many other processes in this representation, a directed graph can be drawn which groups together related systems into new control points and reasonable levels of abstraction. Using the terminology of Chapter 3, some of these processes represent accelerator parameters — they are objects in the OO sense, with interfaces, methods and local data. Mutual dependencies of accelerator parameters are represented by this directed graph.

The orbit parameter must communicate with BPM devices, and these BPM processes (position parameters) in turn each control a BPM driver. The orbit parameter may also contain simulated BPM data or communicate with a simulation that produces such data, encapsulating the simulation and real measurement interfaces within a single object. Another less evident example is a power supply device, which contains an MADC device, a PLC device and a waveform generator (WFG) device, all of which in turn share the same power supply hardware.

The generic parameter DFD of Figure 3.3 contains several processes and data stores, all of which are only moderately self-evident within the context of any specific parameter. The important concept here is that parameters are woven in a web of mutual dependency. The parameter DFD is self-referential, and anything within the "Operate ..." process should itself be described by this diagram down through the lowest levels of the system. This parameter's external interface (as opposed to its functionality) is encapsulated within the "Validate and Sequence" process, which contains the decision-making resources of this process.

Figure 5.1 expands the "Validate and Sequence" process in the generic parameter DFD. Internally, validation and processing of a request to the parameter are handled separately, with validation including type and semantic checks as well as tests for valid limits and parameter constraints. Upon acceptance of a request, it is parsed and new requests are assembled by the "Assemble command list" process.

The vaguest of the processes in this diagram is the process which actually performs commands, and which makes decisions based on events and data received from subordinate processes. This process must handle interrupts (alarms and notifications) and must decide how to perform the commands listed in the data store created by assembly. Because the command list is shared with the "Combine" process, a simple sequencer might never touch

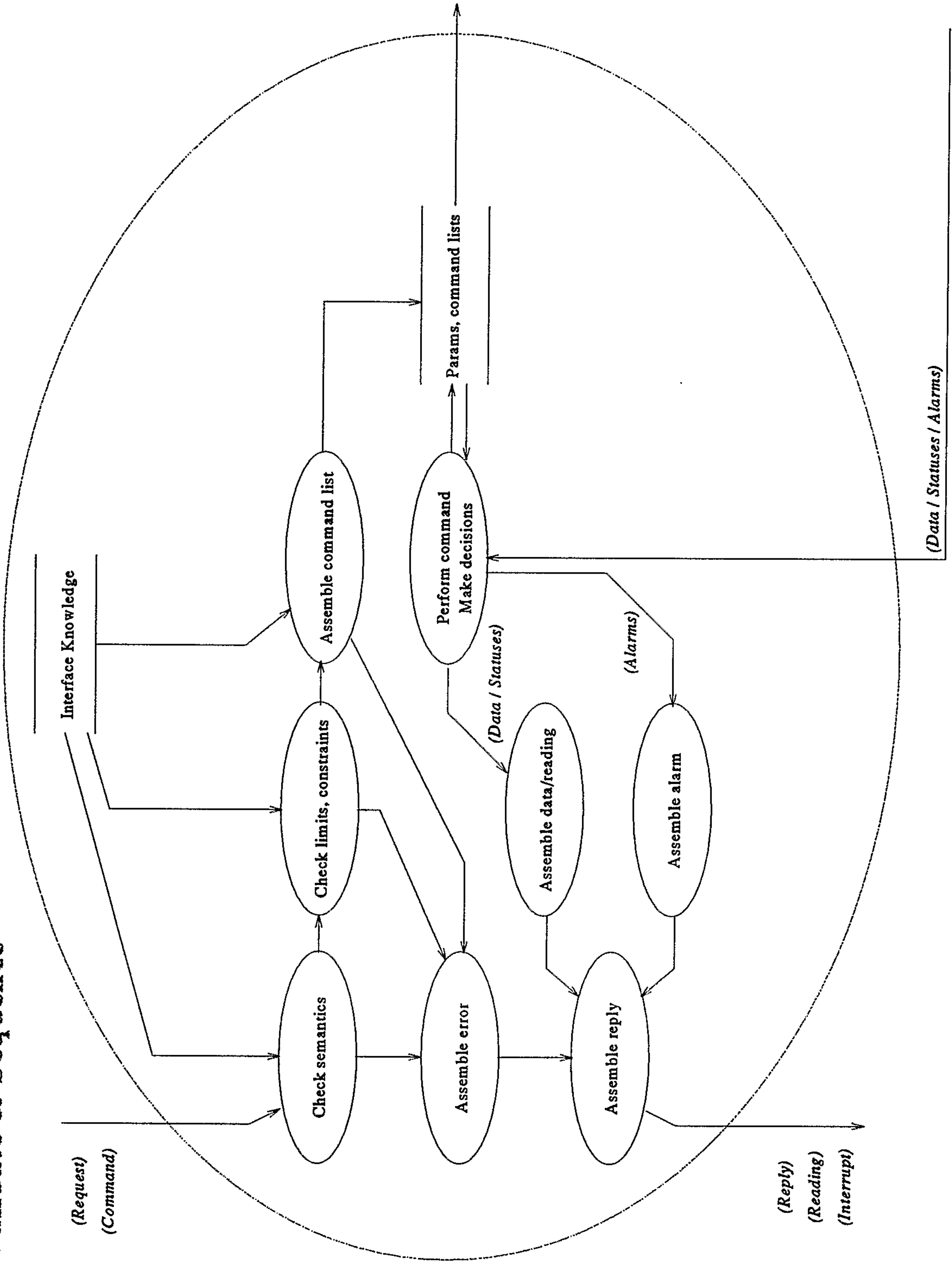

Figure 5.1: A DFD representing the generic sequencing operations in the DFD of the previous figure. Note that the processes of checking and parsing the request are completely separate from the process that performs the request.

the low-level requests, instead only serving as a function that passes returned data upwards. However, the capability exists in this diagram for local feedback; this process may, for example, decide that it can attempt to correct certain error conditions locally (by registering new low-level commands) before notifying its parents and peers.

The "Perform Command and Make Decisions" process can also be wrapped around an application, with the "Params and Command Lists" data store representing the interface to dependent parameters and processes.

There is nothing that precludes the param and command list data store from including data which is relevant only to this level of decision-making. In this sense this system is rather similar to a graphed neural-network. Without appeal to the implications this analogy carries, a simple but powerful realization remains — this model allows decisions and control to be robustly and flexibly placed in their *appropriate* position, without enforcing unnatural hierarchical relationships.

Trims, or small changes in the model that are not well understood, can also be appropriately placed. A straightforward example of this is that the trimming of a magnet transfer function (to translate magnet current to field strengths and vice-versa) is quite different than the trimming of a power supply response, even though both occur in the same chain of parameter dependencies from desired magnetic field to bits sent to a power supply controller.

II. An Application Example: Beam Threading

Beam threading, or beam steering, is an example of a high-level application that must be available during commissioning and routine operations. The basic problem involves three steps: reading the orbit at certain positions (typically beam position monitors), calculating an orbit correction (trim) based on this orbit, and sending the orbit correction back to the machine. A first-turn variant of this process also includes beam loss monitor (BLM) and current monitor readings; there orbit is extended to include any available diagnostic information as to where the beam has gone. A simulation or prediction of the orbit produced by the correction is often included for real-time comparison and analysis.

Here we consider beam threading to be a one-dimensional problem, completely separate from the problem of linear decoupling. Although this is not always the case in practice, it is

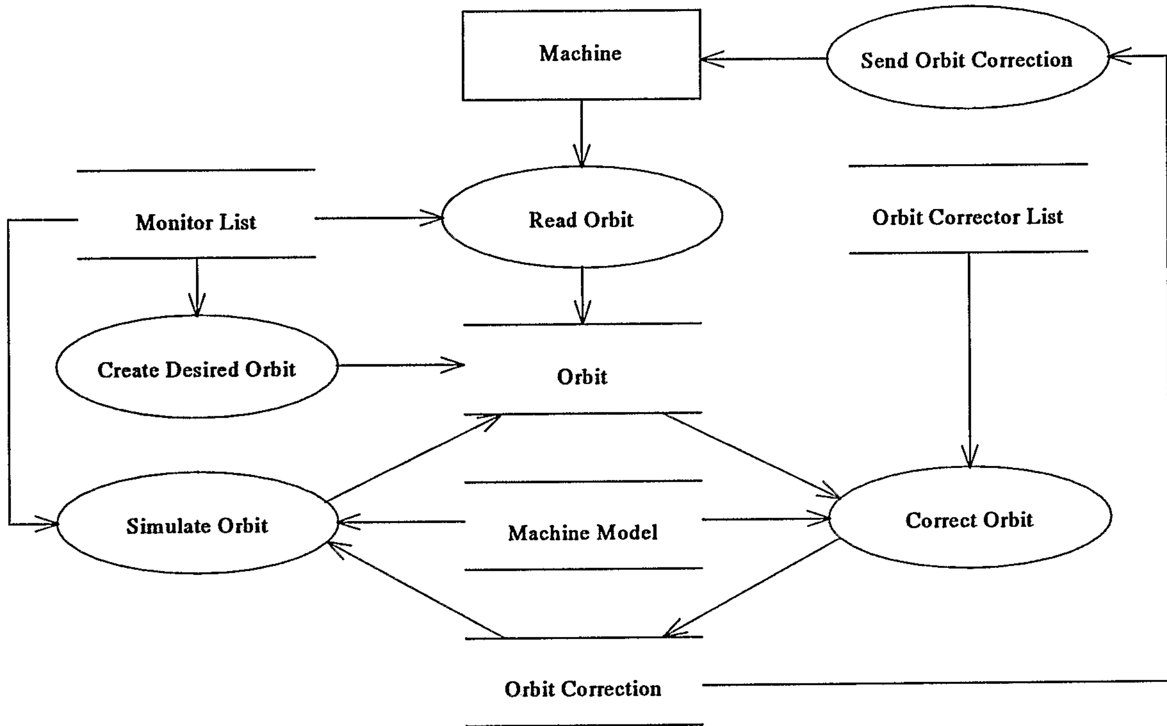

Figure 5.2: Top-level DFD for a simple beam threading application.

a reasonable assumption which leads to two well-understood areas of accelerator operations.

1. The Top-Level Dataflow Diagram

First we describe beam threading in terms of a top-level DFD which defines the scope. Such a DFD is shown in Figure 5.2. There is no "Display" process or actor in this Figure — a passive display does not consume data and therefore is an implicit method of the data object being displayed. External interactions are more an aspect of control flow than data flow and must not unduly influence the data modeling.

There are two processes in the top-level DFD that interface to the actual machine hardware: these processes read an orbit from the machine and send an orbit correction to the machine. These are reasonable processes to separate because their relevant data structures are rather different; however, separating them removes some of their mutual dependence as an orbit construct. Alternatively, one could combine these functions to a single "Manipulate Orbit" process. This distinction is basically one of generalization and reusability; neither way is correct without further context and knowledge of the implementation. Here we choose

the view of modularity and loose coupling, and keep both as separate processes even at this application's top level.

The "Machine" actor is buried inside a parameter hierarchy beneath the "Manipulate Orbit" high-level parameter, as pictured in Figure 5.4. At the lowest levels are hardware components for power supplies, digital readbacks and BPM digitizer cards, while at higher levels some components (position monitors and magnets) are also used by other applications within the control system. Shared resources and distributed permissions management, necessary in our control system, are supported by this design.

At first pass a reasonable "Machine Model" data store is already available in the form of the lattice for the machine[6]. Refinements to this model, and to the orbit simulation process, are currently underway by Waldo MacKay. Physics is done by comparing predictions from well-understood models and measurement, and predictions in this environment hinge strongly on the presence of a *live* model with consistent magnet strengths. This is something that cannot be overlooked for commissioning.

2. Data Structures of Top-Level Data Stores

Of central concern in the top-level DFD are the structures of the "Orbit" and "Orbit Correction" data stores. Encapsulation of an orbit as a stand-alone object is necessary for diagnosis and evaluation of machine condition; orbit *objects* can then be archived and retrieved as necessary. Data store objects and methods must be available to all interested processes in a consistent way. Saving the machine state (including all magnet readings) along with a measured orbit should allow offline as well as online threading analysis.

The structures of "Monitor List" and "Orbit" are intimately related, and an orbit certainly contains a monitor list in its description. Remembering that an orbit is considered to be purely one-dimensional, a one-dimensional instrument list object for RHIC is written in the OM style as

One-D Instrument List
beamline: string plane: int number: int name: string[ ]

The first entry is the name of the beamline that is being used — this name will correspond to a name in a database table in development; it does not necessarily refer only to a beamline as defined in the lattice database. The orbit plane and number of instruments follow, along with an array of instrument names; this array corresponds one-to-one with the name list in another object described below.

There is nothing monitor-specific in this list object; it can just as easily be used to describe a list of correctors. There is also no explicit mention of standard meta-data which should be carried along with every object, such as timestamps and originator.

Additional information about the global state of the machine should be inferred from machine logs and the time stamp. This additional information could include such relevant information as the current machine mode (injection, ramping, flat top) and timing information. The current species could likely be inferred from the time stamp and the run schedule; in practice it will also probably be inferred from the BPM gains.

There is another structure that is part of each orbit, an array of monitor objects. An orbit represents a collection of a variable number of monitors along with a description of the composite itself (the monitor list and other orbit-specific data). This description is consistent with OM techniques, which define ways to describe composite objects. These techniques allow a great deal of flexibility, as the monitor object may be as simple or as complex as necessary. For now it shall be simple:

One-D Position Monitor
name: string status: statusType weight: float position: float gain: float calibration: float

The first three object attributes are in fact relatively generic — many things have a name, a status and a weight. The name here may be either an ADO name or a SiteWide name — the name should also correspond to an entry in the name array in the InstrumentList. This description of a position monitor is generic enough to apply to both BPMs and profile monitors used for position information during beam threading. A full-fledged profile monitor object may also be derived from this object.

Orbit Corrector Lists and Orbit Correction objects follow very much the same strategy, with an Orbit Correction being composed of a set of Orbit Corrector objects and a One-D Instrument List:

Orbit Corrector
name: string status: statusType weight: float angle: float

This is a high-level view of an orbit corrector, with the strength given as an angle. Both the Corrector and One-D Position Monitor objects share the same name, status and weight attributes, so a generalization could be created from which they are derived. For now, though, these object descriptions suffice.

This section would not be complete without commenting on a higher-level issue, the sources of the MonitorList and CorrectorList. These lists of elements are related at the top level in context-dependent ways such as bumps and tuning strategies. As a start these lists could be created directly by the user, choosing BPMs and correctors from lists of elements in the beamline. However, using this approach for routine operations is unreasonable, for "canned" sequences and systematic procedures are necessary for automation and consistency. This application-specific information was called "method data" in Chapter 3.

There are also other issues relating to the sources of the MonitorList and CorrectorList that have to do with the integration and interface of beam threading into the larger scope of operations sequencing. More comments about this are in the last section of this chapter.

3. Top-level Nonhierarchical Processes

The "Create Desired Orbit" process requires a list of BPMs, which it obtains from the MonitorList data store. There are a pair of rather self-evident implementations of this process, either as a user interface actor generating positions for each monitor in the Monitor List, or as a process that fills in a desired position (such as a calibrated zero) for each position monitor.

The "Simulate Orbit" process is only slightly more difficult. An expanded DFD for this process is shown in Figure 5.3. This diagram shows an extra data store, initial conditions for the orbit tracking, and three processes for calculating the tracking parameters (such as cal-

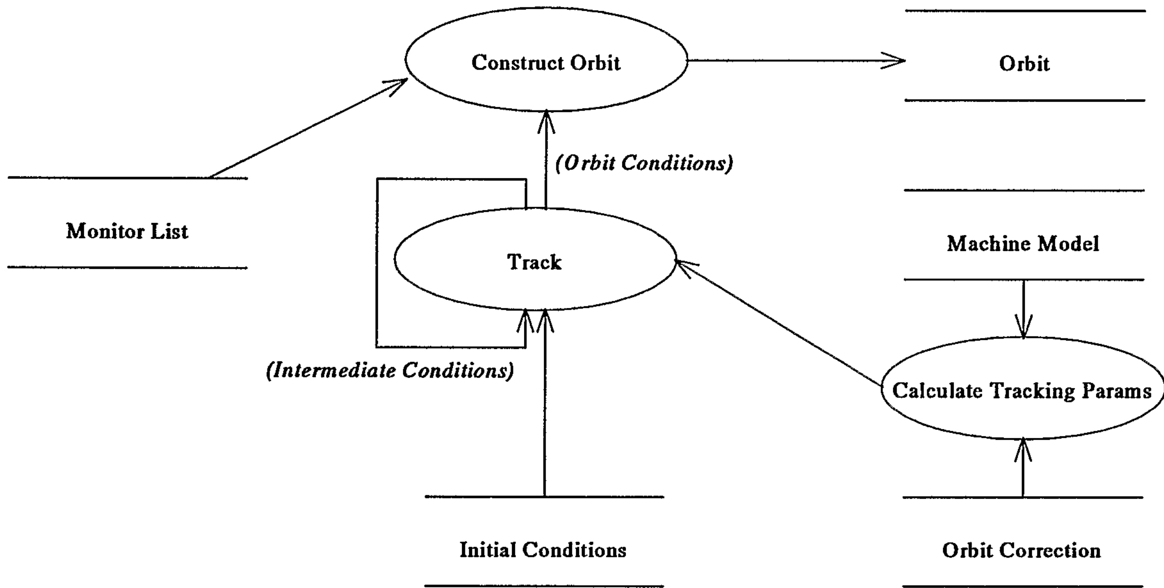

Figure 5.3: Orbit simulation DFD, expanded.

culating linear transfer matrices or products of linear transfer matrices), actually performing the tracking and constructing the orbit based on measurements from the tracking process. Tracking is also iterative: it's output feeds back as new coordinates input into the next round of tracking. These processes themselves can be broken down on other DFDs. This description is consistent with the design of most common tracking programs such as Teapot and MAD.

The "Correct Orbit" process is the most difficult of the top-level nonhierarchical processes. The correction strategy and algorithms are described outside the context of DFDs and OMs in another paper [11].

4. The "Read Orbit" and "Send Orbit Correction" Hierarchical Processes

Implementation: The ADO structure as it currently is defined seems somewhat inconsistent. Controls group members have claimed that ADOs do not intercommunicate, but to encapsulate even a power supply interface in an ADO requires coordination between a PLC, a waveform generator and MADC channels. The MADC interface itself is high-level since MADCs are used for many measurements, which leads to the conclusion that some ADOs manage or coordinate other ADOs.

The "Read Orbit" process in Figure 5.2 produces only measured orbits. An orbit is

considered to be a collection of coordinated BPM readings, with an interface that includes timeline information on when the orbit is to be acquired and which BPMs are to be involved. This orbit object may then be viewed, in controls parlance, as a collection of BPM ADOs, although it is not the only high-level interface to these ADOs. (Individual ADO interfaces are also present, as are memory map ADOs which can conceivably control the same hardware.) The "Send Orbit Correction" process similarly communicates with an assortment of corrector magnet ADOs.

Figure 5.4 shows a hierarchy or directed graph of processes, originating from an orbit process that encapsulates both orbit acquisition and orbit correction. Using terminology from Chapter 3 once more, this process represents the physics parameter "orbit".

Because this orbit process performs orbit correction as well (thus wrapping the beam threading application of Figure 5.2), it must also communicate with many corrector magnets. There is presumably one magnet process for every physical magnet in the machine, since magnets have physical parameters such as measured voltages, thermal tap statuses, etc. The magnet processes in turn coordinate magnet current measurements via MADC and power supply objects, and so forth.

Figure 5.4 may appear to be a hierarchy, but in truth it is instead a directed graph. There is not always a main parent process, and from application to application the control hierarchy changes. Magnets, power supplies and BPMs must report their statuses and readings to several distributed processes within the system. Distributed permission systems, local locking and overrides are critical for such a design.

III. Final Commentary

It is important to realize that the *entire* top-level design of beam threading as pictured in Figure 5.2 can be stuffed in the "Perform command / Make decisions" process of the general "Validate and Sequence" process of Figure 5.1, which is then included in a well-interfaced orbit parameter process at the top of Figure 5.4. The beam threading application thus becomes usable in a packaged and encapsulated form by other applications (including a run sequencer), as well as being able to stand on its own.

To this end it is important that a consistent language for sequencing high-level applications

be designed. Currently we have a package, Glish[7], which allows a great deal of sequencing. Consistency of device wrappers, internal sequencers, error handling and interface design allow clean and integral design of the entire control system.

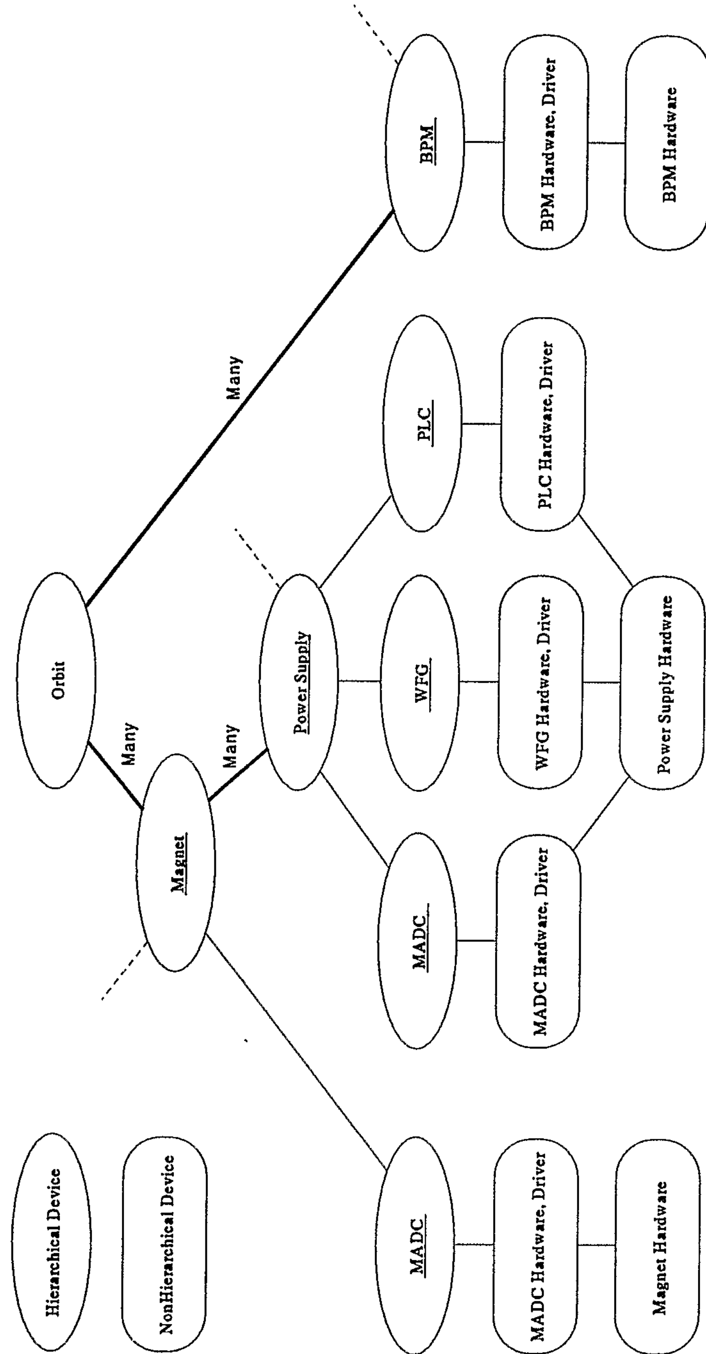

Figure 5.4: The device hierarchy for orbits and beam threading. Ovals represent hierarchical devices (those which fit into the framework of Section I), while lozenges represent nonhierarchical devices. Heavy lines indicate that many subdevices may be controlled by a single higher-level device, and underlined devices are ADOs. Further commentary and details are in the text.

Chapter 6

A Generic Device Description Scheme

I. Motivation and Problems

To efficiently install and control a large system such as an accelerator or transfer line, the relationships between its various elements must be defined in a clear and consistent manner. One particular problem for RHIC, both from an optics/electrical bus viewpoint and a controls viewpoint, has been the relationship between magnets and power supplies — which power supplies control which magnets? This is a trivial question to answer for correctors and trim magnets, but the wire-up issue is nontrivial for complicated bus work such as that for the interaction region quadrupoles. This paper describes a scheme which handles arbitrary wire-up problems with a relational database — this scheme is also shown to be extensible to a general description of design, including data flow in control applications as well as physical installation of complex bus work.

The quadrupole power supply busing for four of the RHIC interaction regions is shown in Figure 6.1, as taken directly from the RHIC design manual[15]. A natural way to view this or any other connection schematic is as a set of boxes (*devices*) with lines (also devices) drawn between them. Devices (magnets, power supplies and busing in this figure) have general attributes such as the name and type of device and the numbers of incoming and outgoing attachments for connections (*spigots*). Connections, the links between the spigots on devices, also have general attributes such as type (a hardware or software connection type). Devices and connections, and their connectivity relationships, can be generally described by entries in a relational database.

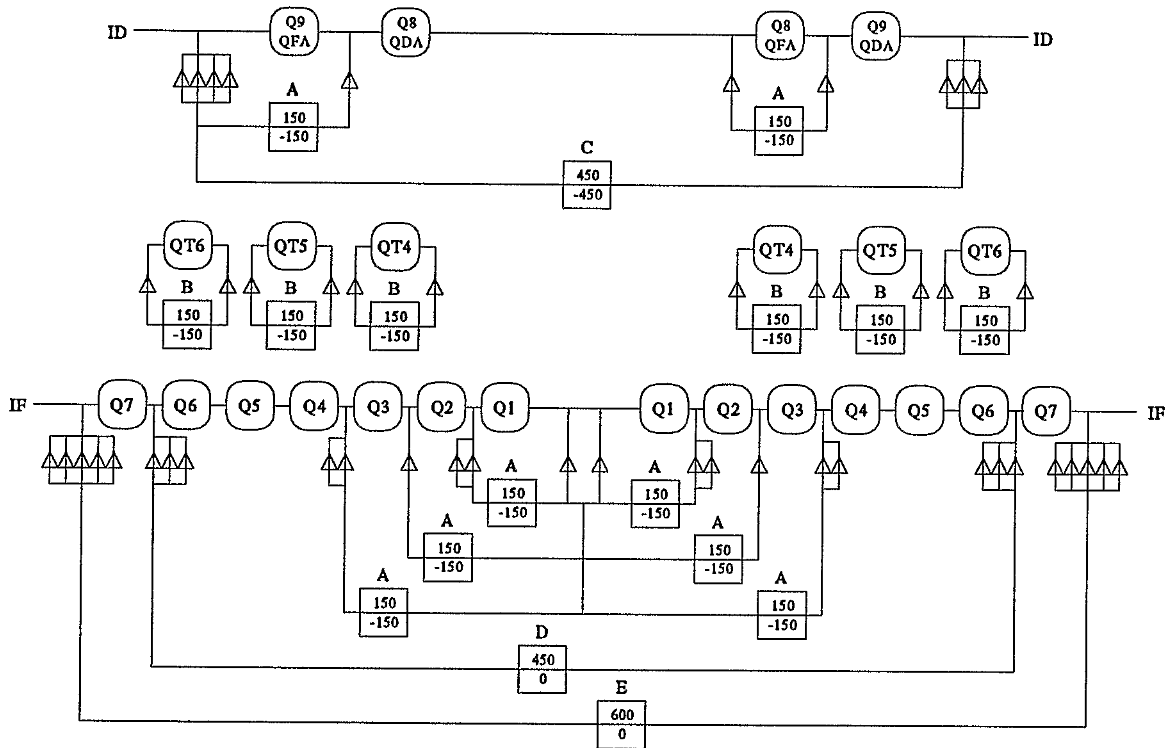

Figure 6.1: Power supply busing for quadrupoles in the RHIC interaction regions (IRs) at 2, 6, 8 and 12 o'clock. Triangles represent 150 amp cryogenic penetrations. This figure is from Bob Lambiase.

The wiring diagram of IR quadrupoles in RHIC, like most of the wiring in most complex systems, is highly regular and duplicitous. There are six identical quad trims in Figure 6.1 (itself applicable to four of the six RHIC IRs), and this trim wiring scheme is duplicated hundreds of times throughout RHIC. A general hierarchical description of this diagram avoids the consistency issues that plague the update of many copies of this information.

To address these issues we have designed a database which can handle arbitrary wire up diagrams, such as that of Figure 6.1. This database is designed to be as flexible as possible, and includes generic templates for common connection schemes.

It must be stressed that these tables are engineered for one specific purpose — to describe connection and containment schemes for generic objects. They are *not* meant to provide a repository for information specific to physical instances of things, and their generality is lost if attributes are added for discrimination of physical instances. Other databases (such as inventory tables) should contain this information, along with references into this database that show how these devices fit into the general wire up scheme. This is demonstrated by examples in later sections.

II. A Fundamentalist Approach to Entity-Relationship (ER) Diagrams

A relational database is comprised of tables, where each table consists of columns (and associated data types) into which data are placed. Data grouped by rows in a table are called table entries.

Magnet Description

Slot SWN*	Serial Name	Type
...	...	...
uq5	ATRQSL008	quad
uq6	ATRQSS013	quad
uq7	ATRQSL007	quad
...	...	...

Power Supply Description

Supply SWN*	Supply Serial Name
...	...
psuq5	ATRPS032
psuq6	ATRPS033
psuq7	ATRPS034
...	...

Power Supply Wireup

Magnet Slot SWN	Power Supply SWN	Polarity
...	...	...
uq5	psuq5	1
uq6	psuq6	1
uq7	psuq7	1
...	...	...

Table 6.1: Three example tables showing magnet busing. The acronym SWN stands for SiteWide Name and an asterisk indicates a primary key. Three rows are shown for each table.

Three simple database tables are shown in Table 6.1. The Magnet Description table has three columns: the Magnet Slot SiteWideName (SWN), which is a unique name for the *lattice position* of the magnet, the Magnet Serial Name, which is a unique identifier for the *physical magnet* which is installed in that slot, and the Magnet Type. The Power Supply Description table includes an SWN and Serial Name for power supplies. The Power Supply Wireup table associates entries in the previous two tables, providing design information on how magnets and power supplies are bused together.

There is sometimes a single column or group of columns in each table which is a "key", a unique identifier for any single table entry, or row. Some tables have no keys, while others may have several which key the same table in different ways. In the Magnet Description and Power Supply Description tables above, the SWN entries are declared as primary keys; during data entry any new entry in a table that duplicates a primary key is automatically rejected.

In the course of database design, tables naturally fall into two categories. One category is for entities or instances of things — magnets, magnet slots, power supplies, wires, cables,

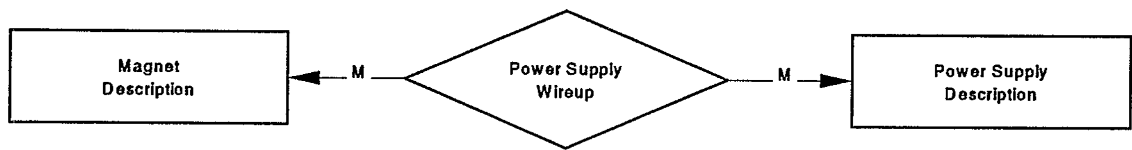

Figure 6.2: An entity-relationship (ER) diagram of the example tables from Table 1.

cards, people and so forth. Tables that describe these instances are called **entity tables.** The other type of table is a **relationship table**, which associates entities, the entries in entity tables.

Entity tables are almost always keyed; they also have other columns which contain descriptive attributes that all entries in the table may share. It is an important and difficult design decision to choose a reasonable level of abstraction for a problem such that the information in entity tables is neither highly duplicitous nor irrelevant. For example, power supplies and magnets share some attributes (color, weight, manufacturer, serial name) but not others (magnet type, magnet half-core serial numbers and power supply limits).

Tables may be related to one another in various ways (hence the term "relational"). A convenient way of diagraming the relational database references between entity and relationship tables is with an **entity-relationship (ER) diagram**[3].

In an ER diagram, entities are represented by rectangles and relationships are represented by rhombi; there is usually a one-one correspondence between these symbols and actual database tables. Directional **arcs** are drawn between entities and relationships to indicate reference, or dependence — in implementation the table at the base of the arc (the table that symbolizes the relationship) contains a column with the same data type as the primary key of the table at the end of the arc. Interpretation of arcs is sometimes simplified by using verbs as labels, which allows one to "read along the arcs". Association qualifiers (such as "M" for many, "alw" for always, etc.) are also used as arc labels. An ER diagram of the tables in Table 6.1 is shown in Figure 6.2.

Tables which have many arcs pointing to them are "fundamental" tables; their entries are referenced, by primary key, in many other tables. Fundamental tables are the first tables filled during data entry. In the next section the DeviceType table is an example of such a table — it contains a list of all possible DeviceTypes for Device table entries. The structure of the tables constructed by the ER method should prevent the entry of a DeviceType in the Device table that is not in the DeviceType table to maintain referential integrity. On the other hand

tables which have many arrows pointing away from them are generally relationships between the various tables to which they point.

The program **erdraw**[3], developed at LBL, was used to implement these tables using ER methods. This program allows graphical editing of ER diagrams, including table attributes and fairly sophisticated delete and update rules. Most importantly, erdraw also produces SQL for table creation, table keying, referential integrity and meta-tables (tables containing descriptions of these tables) that can be read by most database SQL interpreters.

III. The Generic Device Description Tables

Figure 6.3 shows the ER layout of the generic device description (GDD) tables. The six lower tables compose generic instances of devices (including templates of wiring schemes) while the three top tables represent actual instances of devices that fit into these templates. This section describes the generic tables in more detail.

III.1: The fundamental entity tables

Since we seek to represent connection diagrams similar to Figure 6.1, it is reasonable to start with basic entities which represent objects on this diagram. Boxes with external connection points are called Devices and the connection points on devices are called Spigots:

Device: a hierarchical object which contains zero-many other devices and which is contained in zero-many other devices. Devices each also have zero-many "spigots", and have primary-key Name and Comment and Type attributes.

Spigot: an external connection point on a device. Spigots only have directionality within the context on a particular device. The only attribute of Spigots is a primary-key Name.

The majority of arcs in Figure 6.3 (as well as most of the tables) are concerned with many-many relationships between these fundamental entities. A closeup of a quad corrector from Figure 6.1 is shown in Figure 6.4 to clarify this terminology.

There is one more fundamental entity, **DeviceType**, which lists the acceptable entries in the Type attribute of each device. This constraint is shown by the "has" arc between these two tables. Note that this is not an "alw-has" (always-has) arc, so devices may exist with

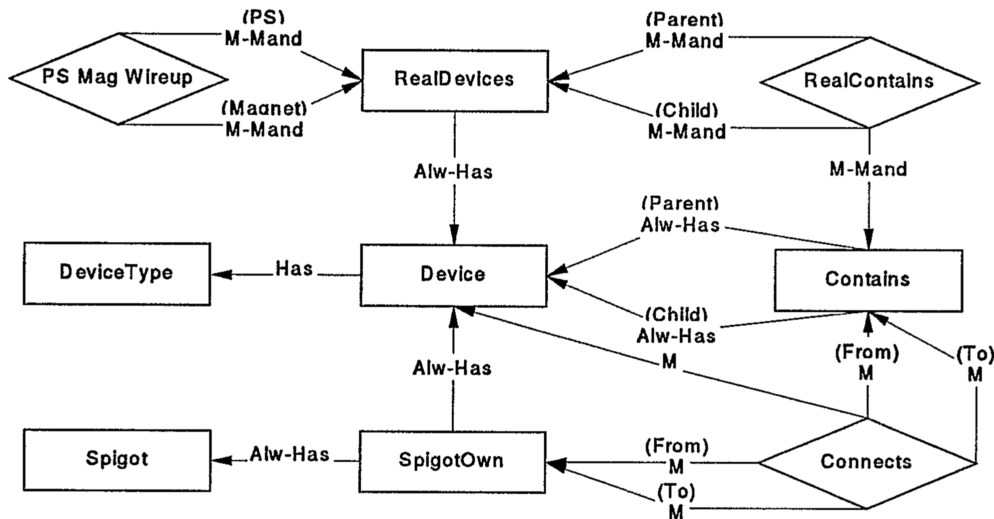

Figure 6.3: The generic device description (GDD) tables in entity-relationship (ER) format. The top three tables are not part of this description, but show the relationship of physical instance tables to GDD tables.

Table Name	Attribute	Type
DeviceType	**Type***	char[20]
Device	**Name***	char[20]
	Purpose	char[60]
Spigot	**Name***	char[20]
SpigotOwn	**Name***	char[20]
	Direction	int
Contains	**Name***	char[20]
Connects		
RealDevices	**Name***	char[20]
RealContains		
PS_Mag_Wireup	**Polarity**	int

Table 6.2: Attributes of the GDD tables. The order listed is the order in which tables should be filled for referential integrity. A star indicates a primary key; attributes in boldface are mandatory for each table entry. Table columns corresponding to arcs in Figure 3 are not included in this list.

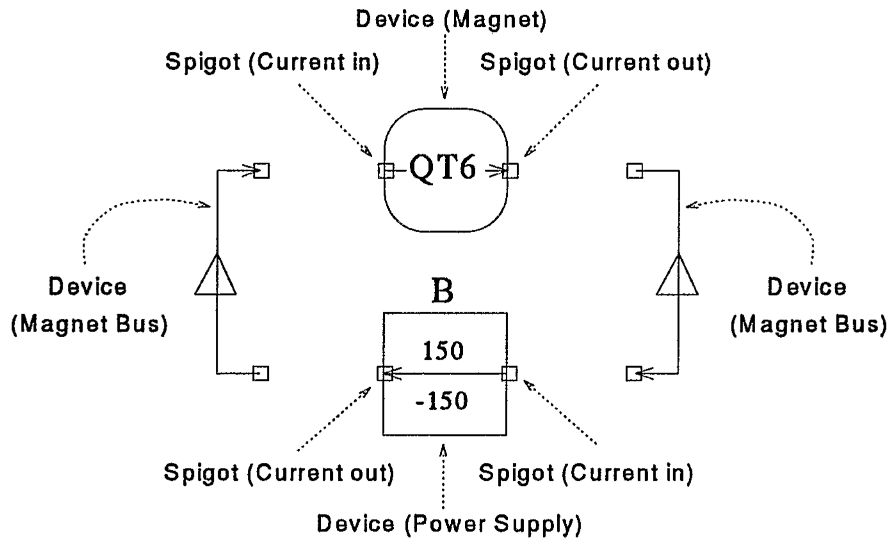

Figure 6.4: A closeup of a corrector magnet bus. Arrows show nominal spigot polarity, and each device has two spigots. The entirety of this diagram is a single generic device template called "1PS-1Mag".

NULL Type or a Type listed in the DeviceType table, but none other.

III.2: Many-many relationship tables

There are two relationships between the fundamental entities Device and Spigot that are evident from the example figure: each device "owns" zero-many spigots and each device contains zero-many devices. Note also that the converse is also true — each spigot is owned by zero-many devices and each device is contained within zero-many devices. More succinctly, there are many-many relationships between the Spigot and Device tables (ownership) and between the Device table and itself (containment). These relationships are represented by the **SpigotOwn** and **Contains** tables in Figure 6.3.

There are interesting things to note about these two tables. First, even though they are many-many relationships, they are entities themselves. For connectivity it is important to be able to distinguish between the same types of spigot on a particular device, as well as the same types of device contained within a larger composite device. Each of these tables must therefore have its own primary key; for lack of better nomenclature this is a Name.

With SpigotOwn the context is also established for directionality. It is clear that on some devices a current spigot is incoming while on others it is outgoing. It is also clear that on devices where this distinction is not immediately apparent (e.g. magnets, ground buses), there

are still assumptions of polarity that warrant this distinction in all spigot-device associations. The SpigotOwn entity has a Direction attribute (±1 or 0, indicating polarity or lack thereof).

III.3: Connections

Circuits are created by attaching spigots together; this is akin to physically performing a connection such as attaching a cable to a socket. Using the device/spigot terminology, a connection is a relationship between a spigot on a contained device (a SpigotOwn entry) and another spigot on another contained device. Both the SpigotOwn entry (specifying a device and spigot on that device) and a Contains entry (specifying which instance of a device within a composite device) are needed for each end of the connection. Different connections may share the same SpigotOwn or Contains references, but not both.

Connections are implemented as the paired many-many relationship **Connects** in Figure 6.3. Here there are two many-many relationships, the *To* pairing and the *From* pairing, which are associated within a composite device. It is also possible (even preferable) to create a Sybase *view* which lists all contained devices and their spigot lists, and associate entries in this view within the Connects relationship — however the ER methodology does not appear to implement this approach.

For a circuit tracing program to work with this data, all circuits must be closed. Internal connections are supported by this framework — if the Contains entry is absent in a Connects table entry, the Spigot listed is presumed to be a spigot on the internal side of the device containing the connection. This will be made clearer in the next section by example.

Every entry in the Connects table can now be interpreted this way: "Within a certain composite Device, there is a connection from SpigotOwn (an instance of a spigot on a device) on Contains (an instance of a device in the composite device) to another SpigotOwn on Contains". Completely general wire-up and connection schemes are supported by this design.

IV. Some Specific Examples

Here we consider two examples. The simple case of corrector and trim magnets is meant to clarify the ER design of the GDD tables. Second, we consider the more complex scenario of IR quad busing as depicted in Figure 6.1; this example also depicts how complex device hierarchies are implemented.

DeviceType Table

Type char[20]
Magnet
Power Supply
Bus
Template

Device Table

Name char[20]	DeviceType char[20]	Purpose char[60]
Magnet1Tap	Magnet	One-tap magnet
Power Supply	Power Supply	One-tap power supply
Magnet Bus	Bus	
1PS-1Mag	Template	Device Template...

Spigot Table

Name char[20]
Current

SpigotOwn Table

Name char[20]	DeviceName char[20]	SpigotName char[20]	Direction int
Mag1Iin	Magnet1Tap	Current	1
Mag1Iout	Magnet1Tap	Current	-1
PSIin	Power Supply	Current	1
PSIout	Power Supply	Current	-1
BusIin	Magnet Bus	Current	1
BusIout	Magnet Bus	Current	-1

Contains Table

Name char[20]	Parent char[20]	Child char[20]
1PS1Mag-Mag	1PS-1Mag	Magnet1Tap
1PS1Mag-PS	1PS-1Mag	Power Supply
1PS1Mag-Bus1	1PS-1Mag	Magnet Bus
1PS1Mag-Bus2	1PS-1Mag	Magnet Bus

Connects Table

Device char[20]	SpigotOwn From char[20]	Contains From char[20]	SpigotOwn To char[20]	Contains To char[20]
1PS-1Mag	Mag1Iout	1PS1Mag-Mag	BusIin	1PS1Mag-Bus1
1PS-1Mag	BusIout	1PS1Mag-Bus1	PSIin	1PS1Mag-PS
1PS-1Mag	PSIout	1PS1Mag-PS	BusIin	1PS1Mag-Bus2
1PS-1Mag	BusIout	1PS1Mag-Bus2	Mag1Iin	1PS1Mag-Mag
Magnet1Tap	Mag1Iin		Mag1Iout	
Power Supply	PSIin		PSIout	
Magnet Bus	BusIin		BusIout	

Table 6.3: GDD table entries for the 1PS-1Mag template.

IV.1: Corrector and Trim Magnets

Consider the simple case of corrector and trim magnet busing, Figure 6.4. This composite device, generically called "1PS-1Mag" here, is duplicated six times in *each* IR quad bus design (Figure 6.1), as well as hundreds of times for correctors and trims throughout ATR and RHIC — Figure 6.4 thus serves as a *template* for this wire-up scheme. Table 6.3 shows the entries in the GDD tables for this diagram.

There are three DeviceTypes in Figure 6.4, a Power Supply, a Magnet and a Bus. The composite generic device 1PS-1Mag representing the entirety of the figure has a DeviceType "Template". There are three other Devices, a one-tap magnet, a one-tap power supply and a magnet bus; the only Spigot necessary is a "Current".

The magnet, power supply and magnet bus each have two Current spigots, in and out. The template here does not have any external currents or control points and thus has no

spigots. The Contains table entries are self-explanatory, but note that there are two different instances of the Magnet Bus device in 1PS-1Mag.

The Connects table first lists the four connections that are obvious, those that attach together the four Devices that make up 1PS-1Mag. The last three connections are internal device connections, and signify that current that comes into the "in" spigots goes out the "out" spigots. This may seem trivial, but when more realistic descriptions are included (external control points for power supplies, voltage and thermal taps on a magnet, etc.) these connections are necessary for a circuit-tracing program to follow current paths within these devices.

This is still a generic representation; 1PS-1Mag is a simple template for wiring which holds for all trim magnets and power supplies of this type. Section VI explains how physical instances of magnets and power supplies relate to the GDD tables.

Another way to implement a trim magnet template is to ignore the buses as uninteresting and simply to join the input and output currents of the magnet and power supply together. This is feasible, and works if that association is all that is needed, but it ignores the fact that the busing is real and has properties of interest itself (such as penetration limits).

IV.2: Complex Bus Work — RHIC IR Quads

It is natural to view the RHIC IR quadrupole busing of Figure 6.1 as a single template that is instantiated four times, once for each of the 2, 6, 8 and 12 o'clock IRs. This template (unlike the 1PS-1Mag template) has four external spigots for the main quadrupole buses. It also quite naturally breaks down into eight smaller templates, six that are closed 1PS-1Mag instances and two that are the main quad focusing and defocusing buses.

The most worrisome aspect of the RHIC IR quad busing is the many-many relationship between power supplies and magnets — most of the quadrupoles are not on a single bus dominated by a single power supply. A closeup is shown in Figure 6.5, showing how this can be implemented in the GDD tables; basically parallel buses are connected to the same input and output spigots on magnet Q8.

A bigger problem arises on the focusing bus, where there are both tees and four-bus junctions. A simple solution is to break the bus between the Q1 magnets into two sections, and to use the junction between them for feed-ins and returns for the six A-type power supplies. The directionality labels on spigots indicate *polarity* of flow, and do not indicate that this is the only direction that current can flow. This being the case, return paths for

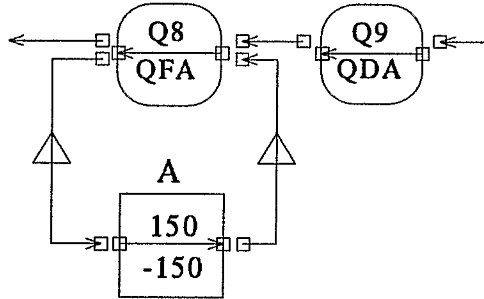

Figure 6.5: A closeup from Figure 1, showing how current tees are implemented.

power supply current are not always those listed — a circuit tracing program using Kirchoff's laws should use these directions only as polarity references.

To describe the entirety of Figure 6.1, eight templates are needed — one for the focusing bus, one for the defocusing bus and six for the quad trim packages. Each quad trim package uses a single magnet and a single power supply, and implies two internal buses. The defocusing bus template requires four magnets and three power supplies, and contains eleven bus connections (two for each power supply and five for the main bus sections). The focusing bus is most complicated; it requires fourteen magnets, eight power supplies and thirty-two bus connections.

V. How to Implement Real Device Instances

Figure 6.6 shows the references into the GDD required to resolve a wire-up scheme for a real trim magnet, in this case `yo4-tq4`, the outer Q4 trim in the yellow ring.

Figure 6.3 showed three tables not in the GDD. **RealDevices** entries are actual devices as referenced by their SWNs or, in the case of Templates, some other unique identifier such as `yo4-tq4-tpl` from Figure 6.6. **RealContains** associates real devices within a real template, and references the GDD Contains table to discriminate separate instances of the same device type in a template. The **PS_Mag_Wireup** table associates magnets and power supplies, with polarities — this table is automatically filled by a wire-up application that uses the GDD tables.

To enter another real trim instance, first the power supply and trim magnet should be entered as real devices in the RealDevices table. The template that represents the "trim package" must also be entered, and these should all refer to appropriate generic descriptions in the GDD Device table. (See bold arrows in Figure 6.6.) The RealContains table should then be filled with a pair of entries denoting where in the template the magnet and power

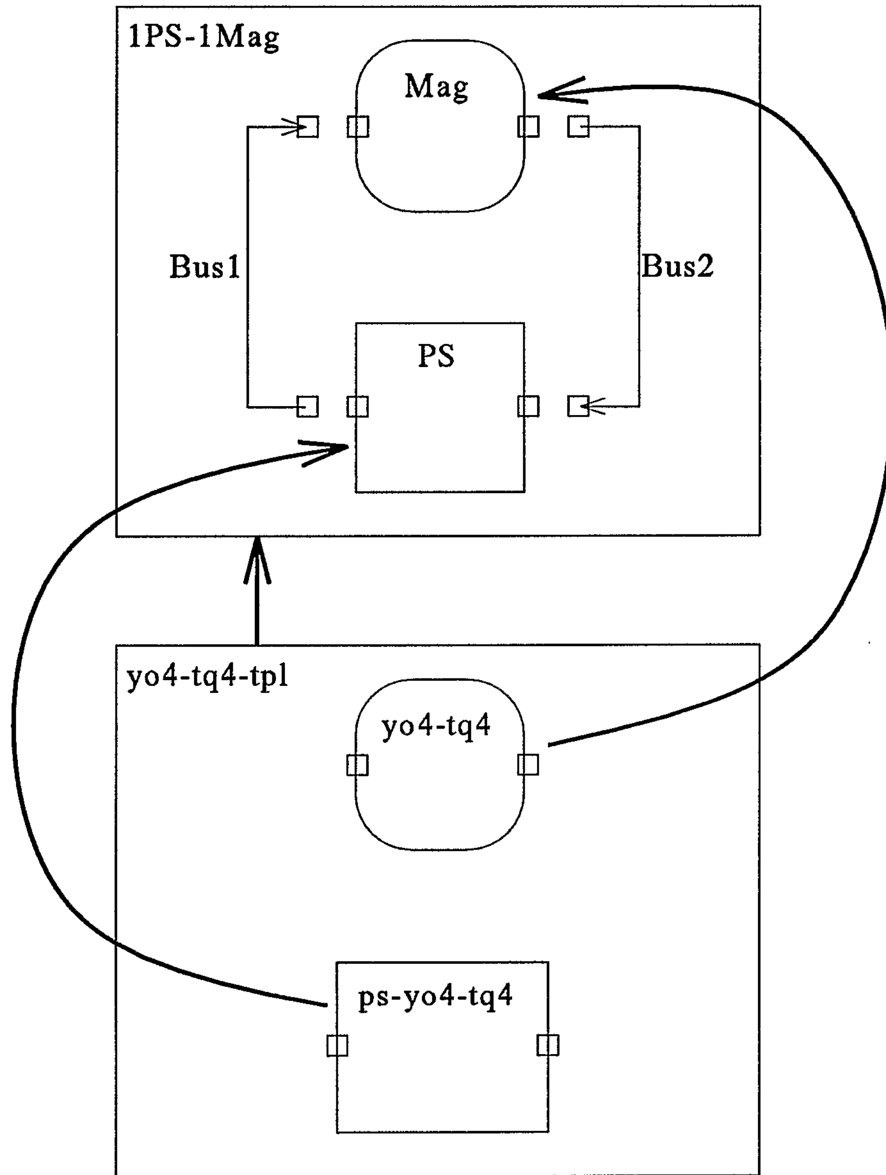

Figure 6.6: GDD references for an actual instance of a generic trim. Note the template `yo4-tq4-tpl`, which ties together the magnet and power supply associations.

supply should go, again with reference into the GDD Contains table.

There are many "generic implications" within a template. A real trim, generically wired and with an "uninteresting" power supply, can be entered as a real magnet and a 1PS-1Mag template instance, with all other elements implied within the template. Above, even the instances of the magnet busing are implied, even though they should be expressed as real instances of busing (with penetration limits, etc) in some other database. **Details of individual elements belong elsewhere** — the associations between them are represented by the GDD tables.

A pair of tables similar to RealDevices and RealContains is sufficient to reference the GDD structure from any database listing physical instances of devices. Examples that are currently under construction include ATR and RHIC magnet busing (physical power supplies and magnets), ATR instrumentation (magnet coil taps, BPMs, etc.) and an instrumentation group cable database.

Templates provide natural encapsulations of connection schemes, just as slots in the accelerator optics database provide natural encapsulations of removable beamline equipment. Templates are an organizational tool, and should not be avoided simply to "streamline" database contents. Also, devices may themselves be templates, and imply other generic structures beneath — this actually means that the number of physical entries of things is *minimized* in this scheme because duplicate implications can reside within the GDD.

VI. Implementation for the AGS to RHIC Transfer Line

To test the viability of this scheme, the magnet busing for all magnets in the ATR was implemented in Sybase tables produced by erdraw from the ER diagram of Figure 6.3. Table creation took less than an hour, and data entry for the 147 magnets in these transfer lines took less than a day.

The vast majority of ATR magnets are individually powered; only the dipoles and lambertsons are on buses which require any templates other than the 1PS-1Mag template described above. The dipoles on these buses are also 4-tap magnets, with two internal coils and two internal return buses. To discriminate between these internal "magnet coil" and "return bus" devices were used. A "current source" internal device was also used to connect the input and output current spigots within power supplies. The 4-tap dipoles, except for xd31 and yd31,

were jumpered internally to look like 2-tap dipoles with a single magnet coil and a single return bus.

The xd31 and yd31 dipoles, and the lambertsons, also have trim power supplies jumpered across their main buses. This situation was handled the same way as multiple magnet buses in RHIC IR quad wire-up, by simply connecting these power supply buses to the dipole and lambertson in parallel with the main arc buses.

A program, **wireup**, goes through the list of all magnet devices in the RealDevice table, and finds the set of top-level templates which contain all these magnets. For each of these templates **wireup** collects information on all internal connections, including recursively traversing other templates, and assembles a list of devices and connections. It singles out the "current source" devices from this list, and traces all circuits that originate at this current source. Both closed circuits (and the magnet coils that reside on them) and open circuits (indicating there is something wrong) are reported. **Wireup** is also smart enough to avoid infinite recursion during template expansion and circuit tracing.

Wireup consists of a 450-line C library of generic routines for traversing the GDD table structure (basically a software implementation of views), and approximately 700 lines of highly recursive C code. When run on the ATR table entries, it produced correct magnet and power supply associations for all magnets in a few seconds of run time. Scaling to RHIC gives an estimated wire-up time of a few minutes.

VII. Concluding Remarks

This chapter has concentrated on describing the GDD and its applicability to magnet busing and wiring schemes. There is, however, nothing specific to this application within the GDD tables. Connectivity diagrams for control flow and for hierarchical relationships between objects within a control system, such as RHIC ADOs[8] and high-level controls hierarchies[11], are quite easily represented within the GDD design. Additional table attributes (such as a ConnectionType in the Connects relationship) can also lead to more application-specific GDD tables.

Chapter 7

User Interface Generalities

We will need a set of tools and libraries that will help us build user interfaces for various applications. The tools should contain most of the items which have proven useful in other applications. Graphical items like menus, scroll bars, pushbuttons, and scrolling lists have proven useful in many different applications on various operating and windowing systems. We would like to use the same kind of graphical items in our applications.

Using the same tools for all of the applications that we write will help us present a consistent interface across the different applications. Choosing an interface that resembles the MS-Windows and Mac interfaces will help the learning curve because most people are already familiar with these interfaces.

When designing an application we should make sure that the user interface and the functionality of the program are separate. That way an application can be controlled by the user interface or by a script without the user interface. Controlling applications with scripts will be necessary for automating complicated tasks.

Plotting standard XY data will be very common, so a tool to do various kinds of XY plots will be needed. This tool should not only output to the screen, but also produce color postscript. That way printing anything that is displayed using this tool will not only possible but trivial. We need to plot flag data, so a 3D plotting tool should also be considered. The plotting tools should accept data in the same format that is used in other parts of the control system. That way any data that is passed around in the system can be plotted easily.

We will also need a tool to display and edit in tabular form any data that is in the control system. This tool should use the same data format that the plotting tools and most other parts of the control system use.

Both the plotting tools and the data editor should have the ability to read their configuration information from an external source. There should also be a naming convention so the

tools can pick up the configuration file based on the name of the data set. If no configuration file exists the tools should choose reasonable defaults on its own.

Currently, we are using the X Windows system with Motif layered on top. We have a set of C++ class libraries that make developing an application with Motif much easier. Motif has the same look and feel as MS-Windows, so many people have no problem learning how to use it. Many professional software packages on UNIX machines use Motif, so this interface should be around for a few years.

We have purchased a table widget that has been very useful for displaying information in a spreadsheet format. We would also like to buy a 2D and 3D graphing widget, so we do not have to build the plotting software from scratch.

Chapter 8

Partial Requirements List for ATR Commissioning

There are two sources for the scope of tasks that need to be accomplished. One is the memo from Mike Harrison[16] that accompanied the document establishing the task force, and the other is from Waldo MacKay[17]. Waldo's list is a bit more detailed in some respects, and we thought it would help to have both definitions of scope preserved here. They do not contradict each other.

To repeat previous discussions in this document, the task force has developed a two fold strategy to design/specify the software for the upcoming beam test. First, a design methodology and a set of tools to document the design have been established. This includes a general device model to guide specific applications. Second, a set of software tools to implement the integrated design. These include data driven graphics tools, database access software as well as event handling and interprocess communication languages.

The design methodology forces us to make a global analysis of the requirements of the control system. Besides identifying many of the pieces of code that will need to be written, this process will also generate the definition of a large set of data stores – many but not all of which may end up as database tables. For example some of the data will remain as configuration files on disk. The ADO RAD files are a significant example of this. The question of what data will reside in the database and what resides elsewhere is an implementation issue, and the design of the system logically comes first. The hope is that the design will lead us to a basic set of software that can be used repeatedly throughout the control system.

With these ideas again in mind we list the things that need to be done. Not all of the items listed below are within the territory of the control system. However, we list them nevertheless. We will indicate by a * (and perhaps some comments below) that particular

items in the lists are in good shape.

The scope as defined by Mike Harrison includes the following. (The three locations referred to are NWA, A-house and 1000P.)

I Accelerator Physics

A. Survey co-ordinates and magnet fiducials for all elements. *

B. Magnet acceptance, sorting if necessary and field quality monitoring.*

C. Application level waveform software for WFG's (ramp page).

D. Beam measurements

  a. beamline lattice *

  b. extracted beam phase space (i.e. beam profile measurements with instrumentation group). *

E. Prototype sequencing code: automatic beamline turn-on and setting.

F. Prototype application code: beamline steering. Includes beamline display with beam position and loss monitor readings. *

II Controls

A. Embryonic console services: graphics, plotting, page display/control, ADO's. Consoles also provided by controls group.

B. Event clock timing system: the master encoder will reside in or around the AGS MCR. Timing signals available in all three locations. The details of what timing signals are needed have not yet been established.

C. Waveform generation and power supply interfaces: available in all three locations with final software at the VME level.

D. Beam Inhibit and Control: no RHIC specific beam inhibit. Beam control provided by the existing AGS system. Note this is **not** the safety system beam inhibit.

E. Network: Ethernet will be available at all three locations. No high bandwidth system available for this test.

F. MADC's: Analog signal read back available at all three locations via standard RHIC MADC's.

G. Digial I/O: Digital device control available at all three locations.

H. Collimator Control: similar system to present AGS control.

Now we reproduce Waldo's list from his talk[17].

I Things to do before beam tests

A. check cooling water on magnets.

B. ramp magnets.

C. check polarities of magnets.

D. pump down line and check vacuum.

E. check interlocks.

F. check other hardware.

1. BPM's: cables and electronics.
2. BLM's (with a radioactive source).
3. Flags: read back pictures with calibration lights.
4. Collimators: check motor control and location read-backs.
5. Current transformers and electronics.
6. Timing system: check signals
   a. to transformers
   b. to BPM's
   c. eventually to injection kicker system

G. Test connection to RHIC abort system.

II With beam ($\sim 10^{10}$ charges of some species, hourly average 1pulse/30sec)

A. Thread beam down the U- and W-lines.

1. Steer the beam onto the flags.
2. Measure the location with the BPM's.
3. Verify magnet and BPM polarities with beam.
4. After reaching a flag with a reasonable trajectory, remove the flag and go on to the next one.

B. Measure the pulse stability from the AGS.

1. Current
2. Position
3. Profile on flags

C. Do fault studies.

1. Check for radiation leaks when the beam hits certain key elements. Of particular interest are:
   a. Access doors, particularly in the split region.

b. Penetrations for ventilation shafts and cables.

c. Thin shielding areas.

d. The top of the berm where Thompson road crosses the beam line.

D. Measure the transverse matrix elements (C, S, C', S') for both x and y.

1. Measure the beam location at all BPM's.
2. Change UTV1 by a small amount and remeasure the trajectory.
3. Reset UTV1 to previous value and remeasure the trajectory.
4. Change UTH2 by a small amount and remeasure the trajectory.
5. Calculate the expected deviations and compare with data.

E. Measure the dispersion elements of the beam line (D, D').

1. Measure the trajectory.
2. Simulate a -0.1% momentum change by ramping all magnets up by 0.1%.
3. Re-measure the trajectory.
4. Calculate the values of D and D' at the BPM locations.
5. Compare with the expected values.

F. Measure the beam shape (hyperellipsoid)

1. Measure the profile at flags UF3, UF4, and UF5
2. Measure the profile at flags WF1, WF2, and WF3
3. Attempt to measure momentum spread with collimator UC1.
4. Calculate emittances, betas, and alphas (horiz and vert) at the flag locations.
5. Measure dispersion of the beam.
   a. Change the momentum of the AGS extracted beam.
   b. Remeasure the trajectory.
   c. Calculate the values of η and η' at the BPM locations.
6. Compare with the expected values.

G. Tune the U-line quads to best match the desired values going into the W-line.

1. Note that the dispersion should be zero at the entrance to the W-line ($20°$ arc).

H. Tune the W-line quads to best match the desired values just upstream of SWM (switch magnet).

I. Scan aperture

Broadly speaking, anything in the above lists which has to do with equipment controlled electronically should in principle be addressable from the control system, in particular, mag-

net power supplies. Closed systems such as vacuum may be excluded although some reporting to the control system might be possible if networks are in place.

Most of the requirements above are associated with high level controls, i.e., software systems that include many front end computers and many pieces of hardware. These are all systems which the people who will actually do the work of building the control system would subject to the global analysis described in this document.

It should also be remembered that the goal of the task force is to prepare an environment which can evolve into a RHIC control system. So we should be careful not to limit the scope of the task force so that we lose sight of this goal. Doing the complete analysis properly is probably the only way to insure against this.

Chapter 9

Summary and Recommendations

This report has described a set of analysis methods that should enable the commissioning physicists and engineers as well as the equipment groups to plan and implement a control system for the accelerator systems of RHIC, beginning with the ATR. Data flow, object analysis, control flow and entity-relationship methods are all necessary elements of the design process developed in this report.

We have proposed a general accelerator device/object model (Chapter 3) and shown by using the beam threading application (Chapter 5) how this model can be used. Implicit in the device model is a hierarchy of levels which proceed from the console level down to the immediate hardware affecting the beam. In between are the front end control hardware and software. Even though we have briefly described this level in the analysis of the beam threading application, it is clear that much more needs to be done regarding the front end analysis. In general the task force has not attempted to analyze the low level hardware domain – a result of the membership of the task force more than the lack of a desire to do so. This is a significant gap in our analysis, and one which we feel must be corrected in the next phase if we are to have a complete understanding of all the components of the control system.

We have also proposed a general language to describe control flow (Chapter 4). This uses a definition of "event" which is somewhat more abstract than usually understood, but we think it is necessary to adopt this definition because it will allow us not only to develop a global understanding of machine and process sequencing, but also permit the ability to simulate these event sequences immediately.

A major concern is organization of data for the control system. We have not discussed a plan for this at length in this report because one cannot know how to organize "all the data" until one has done a proper analysis. There is obviously a lot of data relevant to the

control system currently in existence. Some of this data is discussed in the appendices of this report and much of it is known to one user or another, but it is certainly not clear to a generic user how to go about finding the sources of data at present. Resolving the data organization problem should be a high priority of the next phase. The role (central or not) of the database server should become apparent during this process.

And what is that next phase?, i.e., *What is to be done?* The group of people who will build the control system should now organize for this task. They should prioritize the requirements (Chapter 8) and begin to analyze each of them using the methods discussed in this report. Without strong input and commitment from the hardware groups, much of the analysis will be incomplete, and the operational flexibility that we hope to have in the control system will likely be unavailable. However, we hope that is clear from all the repetition of the word "analysis" throughout this report that we do not expect people to panic and start writing code in the near future. One way to realize the next phase is via the selection of a pre-ops group to continue the work of the task force. This is discussed further in the recommendations below.

With regard to the relationship between the ATR commissioning software that will be written within the next year and the future RHIC control system, a few comments are necessary. The natural tendency after ATR commissioning is completed will be to allow "software inertia" to materialize. However, it should be seen as a normal and natural result of a prototyping test to learn what is not quite right, and the consequences will be that code will be consigned to history. The temptation to fix or build on flawed prototypes must be resisted. Code, after all, represents about 20% of the work of programming so none of it should be considered sacred. So at the end of transfer line commissioning, the operations group should identify the problems with the control system, and dispense with those components.

Recommendations

We will split the recommendations into two groups. Group A deals with the general boundary conditions for organising the future of the continuing task force (CTF). Group B are specifics - *which* techniques, *what* language and so on. Group B is irrelevant unless group A is understood and accepted.

A1. All those affected should understand and agree on the scope of the analysis and design as discussed in the Introduction. The first step is to agree that it is *necessary.* Initially this will involve section leaders from:

RAP
Controls
Instrumentation
Power
Vacuum
Cryogenics
Safety

The less the people in authority trust the analysis and design people, the less the chance of success.

This is the most important recommendation.

A2. People who are writing code need to get used to the idea of *Design* reviews before they start implementing. Such reviews should be coordinated by the CTF, and should use the analysis methods proposed in this report. *Code* reviews on the other hand should be encouraged but are more local - overall coordination will help avoid multiple wheel invention.

A3. Start an operations team. Make them part-time, with

1. A "boss" (from RAP?)
2. Representatives from controls (high- and low-level software, one from hardware = 3?).
3. Representative from instrumentation.
4. ...

They should get together to discuss the direction of the components of the work in the light of commissioning and operational problems. The "boss" should be present at management meetings where decisions are made which may influence ops (this is a **lot** of them).

Don't treat this person as an irrelevancy. When the machine gets closer to its reason for being - producing physics beams - the ops crew will have a difficult job, close to the customer. They deserve to have a voice in decisions which they will have to live with.

A4. Separate analysis from implementation. (Introduction)

A5. Use common languages for analysis which do not demarcate a high/low level boundary in the control software. (Introduction and chapter 2)

A6. Control flow should be modeled throughout the software chain – again, no high/low level demarcation. (Introduction, pages 4-5 and Chapter 4)

A7. The control software should promote a better understanding of the machine physics rather than to just provide a set of knobs for twiddling. Physics modeling should be an integral part of the operation, not an optional and esoteric tool. (Chapter 3)

A8. The CTF should prioritize ATR operational requirements. (Chapter 8)

A9. The user interface should be detachable from applications. (Chapter 7)

B1. For common analysis languages we propose using the Object Modeling Technique (OMT), Extended Entity Relationship (EER) and glish for their respective domains. (Introduction, page 4 and Chapters 2 and 6)

B2. Control flow should be analysed as in terms of an event analysis (where event is understood to be a structure including name, type, value, sources, consumers, definition...) as discussed in Chapter 4.

B3. Analysis of devices and their connections should use the "generic device definition" (gdd) structures (Chapter 6).

B4. Management should encourage the various RHIC groups to store data (survey, inventory, configuration, etc.) in the central SYBASE database and manage this information using the available database entry and retrieval tools provided for PCs and workstations.

B5. The data entry for the ADO CDB and SWNameADO table should start as the ADOs get defined and deployed to the FECs. (Appendix B)

B6. The ADO parameter data initialization files should be be filled as the ADOs are deployed to the FECs. (Appendix B)

B7. We recommend the immediate purchase of the **PowerBuilder** SQL Front End tools for PC/Mac/UNIX workstations. (Appendix C)

B8. The user interface will primarily be based upon the UITools layered on top of OSF Motif. But we expect TCL/TK, LabView and even pure X based tools will also be used, at least initially.

Appendix A

Accelerator Configuration

The configuration of the machine that we are going to build is stored in various databases. The optics database holds the design of the machines that will be built. The optics configuration for the transfer line from the AGS to RHIC, denoted as ATR, and RHIC itself are stored in separate databases. The Generic Device database, denoted as GDDB, is also required for design information of the individual devices that are needed. This configuration consists of:

1. The order of placement of these elements in the line or ring.
2. The type of devices, such as dipoles, quadrupoles, etc.
3. Properties of each device, such as aperture, current limits, expected operating strengths, survey information, etc

There is a lot of information here that is not always easily accessible in a simple form. Furthermore, different groups want the information distributed in different forms, such as:

1. The survey group wants the survey information of the elements and their corresponding fiducials.
2. The simulation group wants the information in a form for existing tracking programs.
3. Other groups have their own specialized needs.

In order to overcome these difficulties, we have setup a directory called Holy_Lattice which contains all the configuration information listed above and more. Within the Holy_Lattice are four separate directories for the four different beam lines required. The two rings for RHIC are put into Yellow (counterclockwise ring) and Blue (clockwise ring) directories respectively. The ATR for injecting into the above rings are in YTransfer (for Yellow ring) and BTransfer (for

Blue ring) directories. Fig. A.1 gives a detailed schematic of the files found inside Blue and Yellow directories, while Fig. A.2 shows the schematic for the YTransfer and BTransfer. In these figures, oval boxes are for processes and square boxes represent data files. Additionally, the lines coming from the files to the processes show what information is used by the process. Other information in the figure indicates the file system location of the process (in []), such as one of the LAMBDA[6] suite of codes and () indicates the data representation.

The most important data deriving from the accelerator configuration data for the control system is the *NameLookup* table (found as a SDS data set in the Holy_Lattice directories with the name, *Namespace*. The structure of *NameLookup* is defined below as:

```
struct NameLookup {
  int lattice_index;
  int atom_index;
  int fid_index;
  int network_index;
  int type;
int orientation;
char Machine[4];
char InOut[2];
char Section[3];
char DeviceName[8];
short DevNo;
  char SiteWideName[20];
char SurveyName[16];
char SerialName[20];
char LatticeName[20];
char GenericName[20];
  int CoordinateType;
  double Scoord;
  double Sequiv;
  double Ncoord;
  double Wcoord;
  double Ecoord;
  double theta;
  double phi;
```

```
    double psi;
  };
```

Each device has its own data page in this structure. The definition of each element is defined as follows:

lattice_index The index is an integer pointer to the device in the Lattice file, a -1 means no corresponding device in the Lattice file.

atom_index The index is an integer pointer to the device in the Twiss or Survey file, a -1 means no corresponding device in the Twiss or Survey file.

fid_index The index is an integer pointer to the device in the Fiducials file, a -1 means no corresponding device in the Fiducials file. This is not used for ATR data.

network_index The index is an integer pointer to the control computer network.

type Is an integer referring to the type of device, such as a quadrupole, dipole, etc. These types are defined in the header file 'names_devicetypes.h' located in /rap/horst/include.

orientation Is +1 if the downstream clockwise end is the lead end, else it is -1.

Machine Is a three character abbreviation for an ATR or a RHIC beamline.

InOut Is either 'i' for inner or 'o' for outer ring.

Section Is a two character designation for logical sections in RHIC or ATR.

DeviceName Is a short mnemonic that gives a clue as to the type of device that is referred to.

DevNo Is an integer used to differentiate similar devices in the same section.

SiteWideName Is a unique name given to every device in the RHIC - ATR complex. It effectively associates a location to each device. This name may be derived from other fields. See Appendix D on Naming Conventions.

SurveyName Is the name of the position used by the survey group for placing the device. This name is the root of the fiducial name used in the Fiducials file.

SerialName Is the name of the actual device placed at this position.

LatticeName Is the name of the device used in the Lattice file.

GenericName Is a model name of the of the device that will be placed in this position.

CoordinateType Refers to three kinds of coordinate types; Traj for trajectory; IP for Intersection point of upstream and downstream rays and Mech for mechanical center.

The following are the device coordinates based on the MAD coordinate system.[14].

Scoord The longitudinal position of the beam through the line. Note, this is clockwise for both Blue and Yellow rings.

Sequiv A longitudinal position along the beam direction that associates a common location to corresponding points in the Blue or Yellow rings. This problem arises because the longitudinal coordinates of the two rings differ since the rings are at different radii except at the IP points.

Ncoord The north coordinate.

Wcoord The height coordinate.

Ecoord The east coordinate.

theta The azimuthal angle measured clockwise from Ecoord, looking down.

phi The pitch angle.

psi The roll angle.

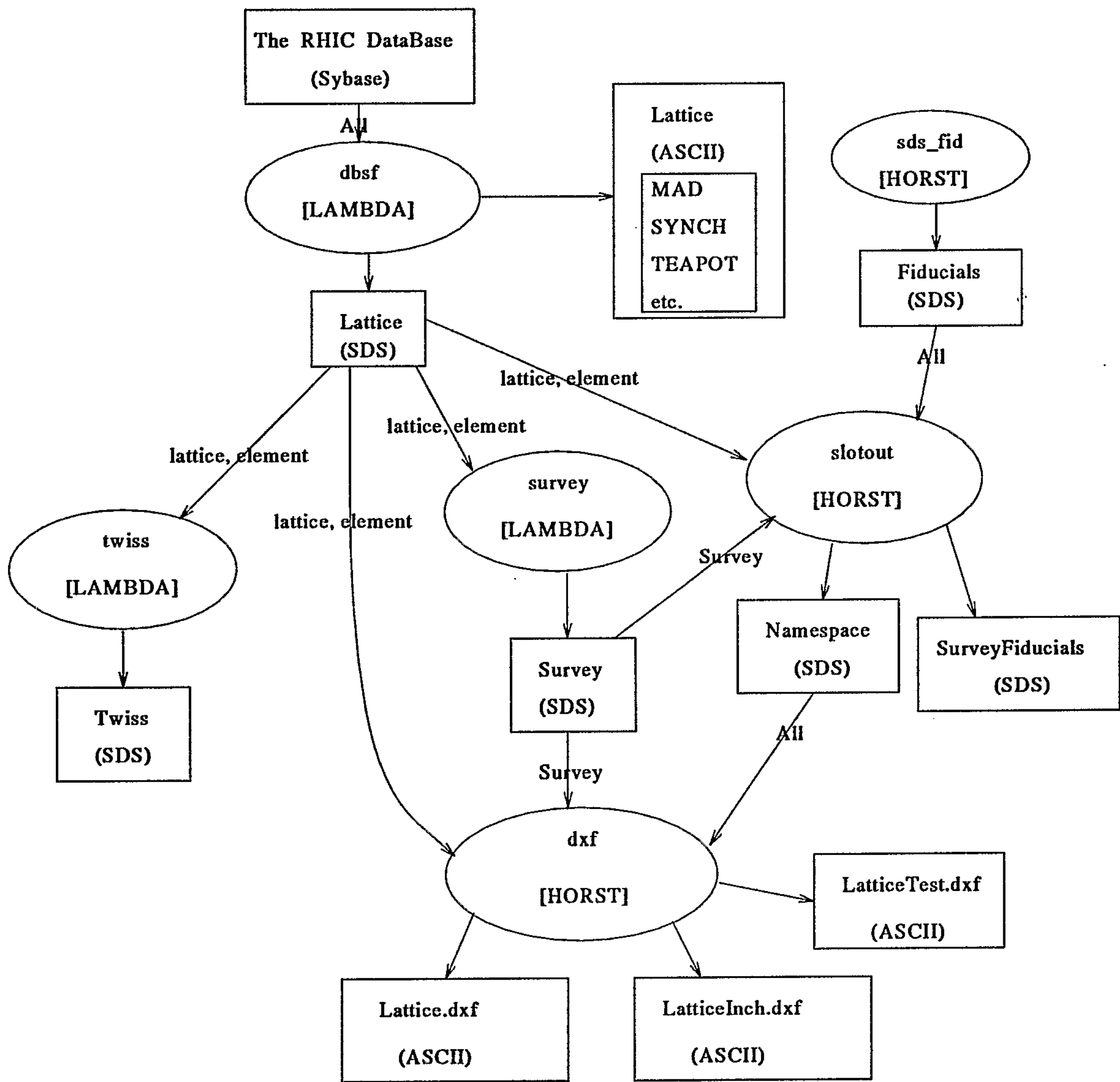

Figure A.1: Data Flow Diagram for RHIC Physics Design

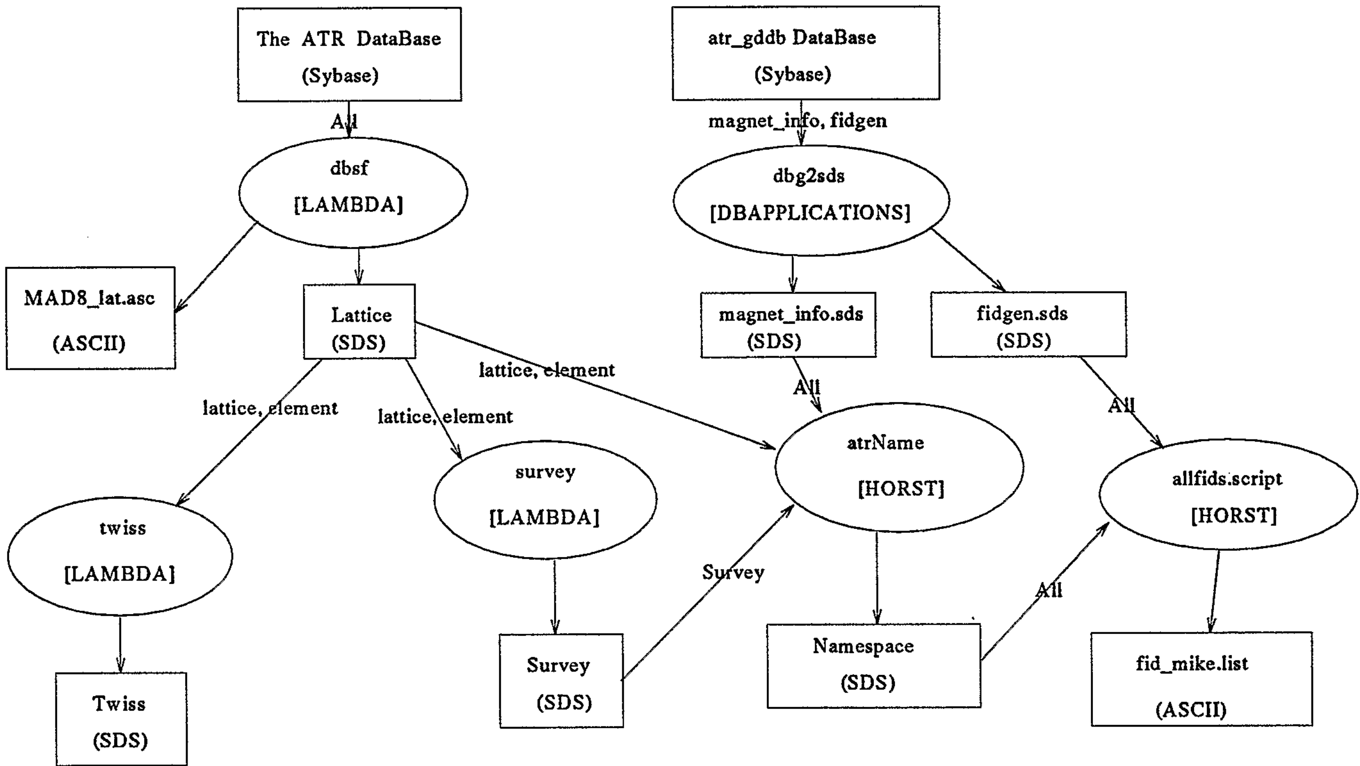

Figure A.2: Data Flow Diagram for ATR Physics Design

Appendix B

ADO Configuration Data Base

The ADO configuration database plays an important role in organizing the ADO related data. ADO_CDB is a repository for all the ADO classes, their instances and ADO Parameters for the RHIC Control System. The applications use ADO_CDB in order to write the programs that need to access ADOs that exist on various Front End Computers. (refer to Data Flow Diagram for ADO_CDB).

ADO_CDB is linked to the NameLookup table via another table called *SWNameAdo*. Using this table, one can relate a **SiteWideName** to the ADO instances that are present in the ADO_CDB. Once the ADOs are known, one can use the ADO_CDB to get more information about these ADOs and their Parameters.

Data Input for the ADO_CDB

ADO Class Data input:

The ADO designer creates a RHIC ADO Definition(RAD) File. The RAD file contains the definition of the ADO Class in terms of data and its methods. This file is used to build ADO header files and the C++ source files. This software is then loaded into the Front End Computers. The ADOClass table and Parameter tables in the Configuration database are filled using the same data as in the RAD files. This assures the data integrity between the FEC ADOs and the ADO_CDB.

The user writes the initialization values for the given ADO class parameters in a file with a name ADO Class Name followed by ".init".

ADO Instance Data input:

The FEC application designer enters the ADO instance names in the ADO_CDB and also enters ADO instance related information in the *ADOInstance* table.

The user writes the initialization values for the given ADO instance parameters in a file with a name ADO Instance Name followed by ".init".

Status of the ADO_CDB

Currently the ADO_CDB has been designed and resides on the RHIC Sybase server. Some work related to the FEC Events that will reflect in this database is underway. Events are described in more detail in Chapter 4. Currently the data is filled in manually and displayed using Sybase supplied tools such as *dwb* and *isql* on Unix workstations. More user friendly packages are being investigated for the ease of data visualization on both the Unix workstations and PCs. The description of such tools can be obtained in the appendix on **Commercial SQL Front-End Tools.**

There are several documents describing details of the ADO classes. They can be found in [8][9][10].

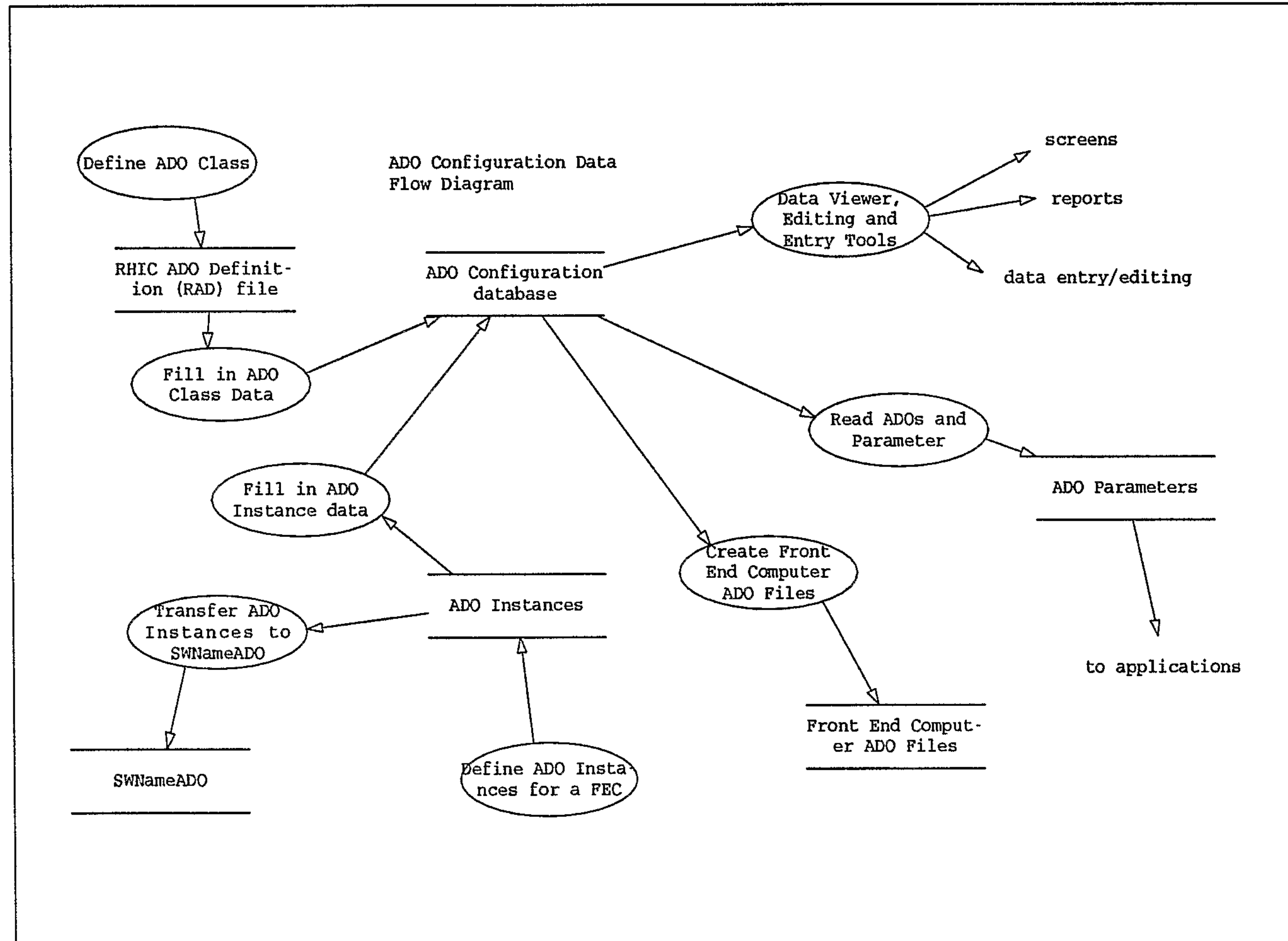

Figure B.1: ADO Configuation Data Flow Diagram

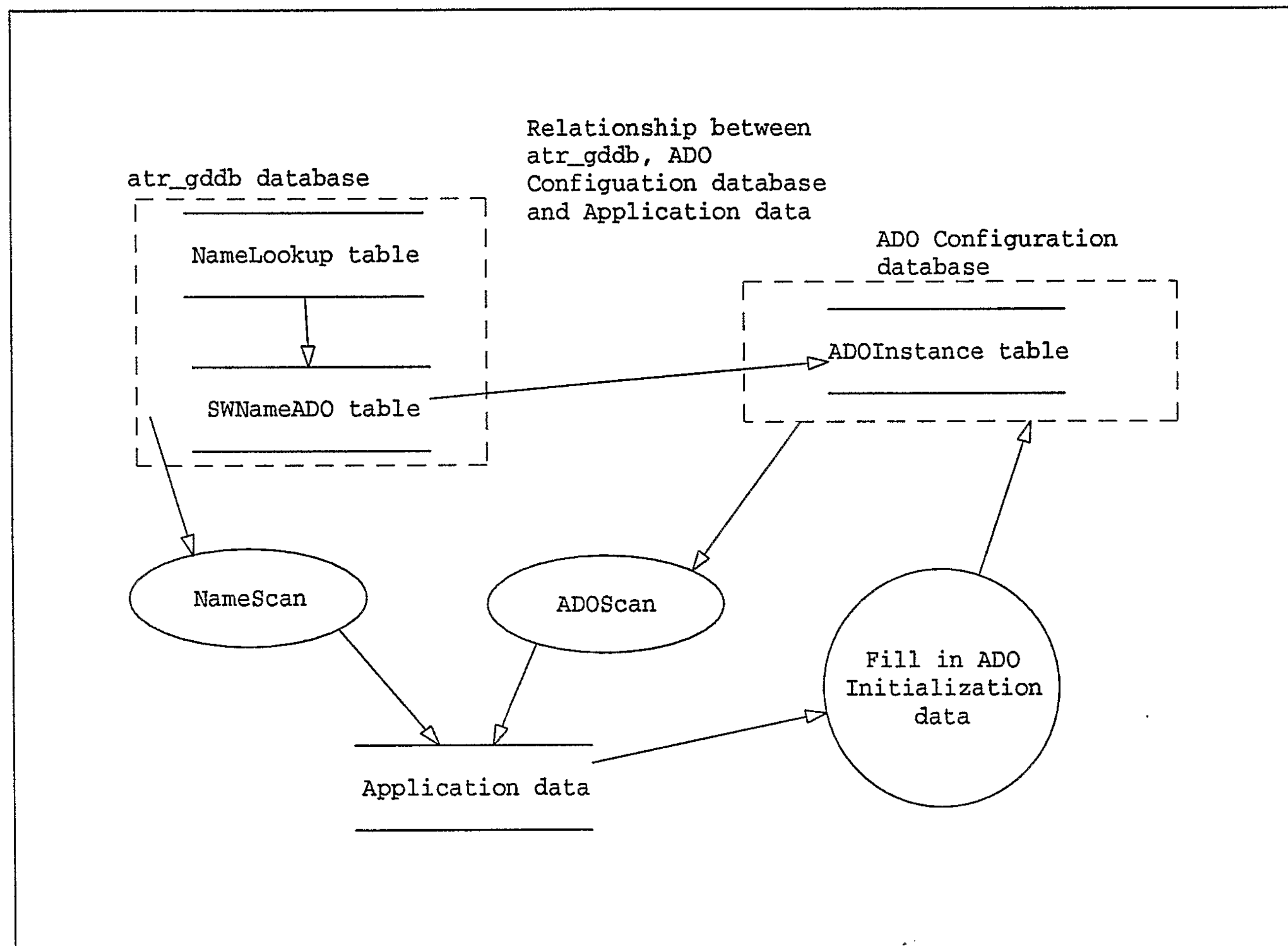

Figure B.2: Relationship between ADO, gdd and Applications

Appendix C

Commercial SQL Front-End Tools

A relational database server from Sybase has been purchased to provide a central data repository for a variety of groups related to the RHIC project. Information related to instrumentation, controls, vacuum, and physics among others will be stored in this database and will be made available to any interested parties.

In order for this effort to be successful, we need to make sure that 1) the data gets entered into the Sybase server, and 2) the required data can be retrieved from the server. Unfortunately, the tools that come with Sybase for entering and retrieving data are based on SQL entered onto a command-line or importing/exporting from files and are, therefore, only useful for experienced users.

This is not a new problem with SQL databases and many commercial companies have stepped in to provide software to "front-end" an SQL server. This software provides GUI-style (point and click menus/buttons) interfaces for entering, editing, and retrieving data from SQL databases. The intention here is to make access to an SQL database as easy as possible so that people are willing to both get their data into the database and go to the database to retrieve information.

Based on the expected usage of these tools for the RHIC project, and the available monetary resources and personnel, the following requirements have been set for the purchase of commercial SQL front-end tools:

1. The tools should allow an experienced database user/software developer the ability to create an easy to use front-end that another group can use for entering information into the database. These front-ends will be somewhat customized to the particular data being entered/edited. The developer will develop the front-end in consultation with the group primarily responsible for the data. Specialized queries and reports may also be

required or the users may use the available generic querying methods (see below).

2. The tools should allow an occasional user an easy to use (graphical, Non-SQL), generic method for querying the database in order to obtain a listing of the data of interest. This may require that special "views" of the data be created by a database administrator. However, once created, the user should be able to perform queries on these views just like any other database table. The user should be able to print the data listings created using this generic tool or to export the data to a file.

3. The tools should run across multiple computer platforms in such a way that a front-end created on one platform can be moved and ready to run on another platform in less than an hour. The platforms supported should be at least DOS/Windows, Mac OS, and UNIX. The UNIX support should include at least support for SUN Solaris with SGI-IRIX, HP-UX, and/or IBM-AIX a plus.

4. The cost of the developer tools for all three platforms should be less than $10k and allow for the distribution of front-end applications free of royalities or run-time fees. Tools for generic querying of the database should cost no more than $200 per seat, with $100 a more acceptable target.

An evaluation of the available commercial packages has taken place over two months at the end of 1994[18]. This evaluation was made in conjuction with a BNL committee set up to evaluate and make recommendations to lab personnel about such products. Of the many SQL front-end products available, two were selected which met the above four requirements: Omnis 7, version 3.0 from Blyth Software and PowerBuilder, version 3.0a from PowerSoft Corporation.

Although it was felt that both products would serve the purposes outlined above, PowerBuilder has the most impressive set of tools, good third party support and a great reason to remain close to Sybase changes (PowerSoft was recently acquired by Sybase). We recommend the immediate purchase of PowerBuilder for Microsoft Windows ($3k) and the Mac/UNIX versions when they appear in early 1995.

Appendix D

Naming Convention for RHIC/ATR Devices and Signals

The naming convention for RHIC and ATR has its origin in the RHIC Design Manual[13]. This appendix will define the syntax for the convention and list some examples so that you should be able to construct names for devices. A centralized collection of device mnemonics and names is being maintained by G. Trahern. If you have questions regarding the assignment of a name to a device, or would like to add new devices to the list, please discuss with him.

The character string for a complete device name, the so-called **SiteWideName**, differs between the ATR and RHIC. This reflects both the historical practice of the two design groups as well as the obvious differences between a transfer line and a circular accelerator. However, the naming *convention* which specifies what information is needed to make a **SiteWideName** is similar for the two machine systems.

The goal of a naming convention that spans both the ATR and RHIC is a method to name equipment not only for installation purposes but also for the control system environment. Because console programs in the past have had restrictions on the numbers of characters that could be displayed easily, one was forced to abbreviate device names. Currently it is unclear if there should be such restrictions on the way one chooses to address a device via a console application. It should be an implementation detail whether one types the name in or uses the "point and click" route. A modern console application could use an arbitrary length string without much difficulty as long as the data is described properly. Unfortunately, it became clear from conversations and a study of what was already in place when this convention began to solidify that people *really* want abbreviations for names of devices if only to make labels on drawings accessible. So we have surrendered to this demand. Presumably most of the more common device abbreviations will become memorable over time, and a dictionary should be

available in the operator console in case of confusion. A list of device abbreviations and their definitions will be made available periodically.

Signals are not independent of the devices that produce and receive them. And neither are the signal names. However, an ambiguity often arises with regard to which device to associate a given signal. For example, naming the current generated by a magnet power supply might seem a straightforward assignment. But what of the magnet current itself supported by that power supply? Do we associate the current with the magnet or with the power supply? A current measurement device on the magnet's lead wire is presumably the device directly measuring the magnet current. Is that device independently named from the magnet, or is it a subdevice of the magnet? And again, is the signal associated with the magnet or the sensor?

These issues require choices. The choice we will make is, as much as possible, to associate signals with the device closest to the beam. So in the above examples, the magnet current is associated with the magnet through a sensor which is a sub-device of that magnet. One can talk of the magnet current or the reading of the sensor defined as a sub-device of the magnet. If one thinks about this problem in detail, it will be clear that there are remaining ambiguities in the assignment of signal names. Resolution of these assignments will require more analysis of actual systems before a complete solution is reached.

However one assigns a signal to a device, the name of a signal will include the device name as part of its full name. There should also be a clean separation between the device part and the signal part of the complete name and consequently we will separate the two using a colon, :, i.e,

complete **Signal name = DevicePart:SignalPart**

where **DevicePart** is the **SiteWideName** for the device, and **SignalPart** is the specific signal name, e.g., I for current or V for voltage.

The naming convention is *not* case sensitive. Combinations of lower and upper case are not distinguished from names with either all upper case or all lower case. Do not expect to be able to differentiate names according to upper or lower case.

Convention Syntax for DevicePart

The general syntax for the **DevicePart** of a name is as follows. One should think of the device name as composed of several components. How these component are put together in

a character string to form the **SiteWideName** for the device will differ in the ATR and RHIC. But the components are similar. They are:

Machine, *Section*, *DeviceName* and *DeviceNumber*.

Examples of *Machine* are **ATR**, **B** for the Blue ring and **Y** for the Yellow ring. Examples of *Section* are **U**, **V**, **W**, **X** and **Y** for ATR sections, and the clock numbers **1-12** for RHIC. The *DeviceName* is an appropriate/acceptable abbreviation for the device and will be discussed in detail below. The *DeviceNumber* is assigned differently for ATR and RHIC. In the ATR a sequence number is assigned to each device from **1-N** where N is the last instance of that device type in a section of the ATR. In RHIC the *DeviceNumber* is the half cell number for a section(**1-12**) of the ring. The half cell number runs from 0 to 20 or 21 starting from 0 at each of the six interaction points (IP) counting clockwise and counter-clockwise from the IP. The ambiguity of assigning 20 or 21 occurs at the center of each arc where the half cell number 21 is assigned to the odd numbered sections (1,3,5,7,9,11). Both rings in RHIC are numbered in the same way, i.e., the beam direction is irrelevant to the sequence of naming. Finally, and this applies to both the transfer line and RHIC, if there is more than one occurrence of a device in a particular subdivsion, e.g. a half cell in RHIC, then the device number is incremented by a decimal. For example if there are two beam loss monitors in half cell 4, then they would be distinguished as 4.1, 4.2.

(For historical reasons the full RHIC name has another component not present in the ATR name. This is the designation of Inner or Outer, **I/O**, for each section of the ring. This is obviously important information for RHIC installation, and is included as part of the full name. We did not include it as part of the general syntax above since we are trying to stress the commonality of the convention for ATR and RHIC. But the full name of a RHIC device includes this designator as well.)

Given these components to a name, how does one actually spell it? Read on....

ATR examples

For ATR, one just concatenates (no spaces) the Section, DeviceName and DeviceNumber components, e.g.,

ATR **SiteWideName = SectionDeviceNameDeviceNumber**

For example, consider a device such as a dipole. The device mnemonic for a dipole is D(d), either lower or upper case. In the ATR one will have a name like UD1(ud1) which is the first dipole in the U line. That's all there is to it for the ATR. Of course, there are exceptions to the stated rule. But these exceptions are rather few. These exceptions usually

occur at points where the device is common to two distinct beamline sections. For example, the SiteWideName "swm" (for the switching magnet that diverts beam from the W line to the X and Y lines) is one of the more prominent exceptions.

RHIC examples

For RHIC, all five components of the name are used.

RHIC **SiteWideName = MachineISection-DeviceNameDeviceNumber**

or

RHIC **SiteWideName = MachineOSection-DeviceNameDeviceNumber**

depending on whether one is dealing with the Inner or Outer parts of a ring.

So for the dipole in RHIC, one would have a name like BI5-D8(bi5-d8) which means the dipole in the Blue, Inner ring, in half cell 8 of section 5.

Device Mnemonics

There is nothing more arbitrary than an abbreviation for a device. It is impossible to get everyone to agree. So we have chosen to use mnemonics that have been in common usage at BNL where possible. Some of these abbreviations are so brief that we doubt if anything will aid the memory but frequent usage. In other case the abbreviation is fairly obvious. A good working rule for device abbreviations is to remove the vowels from the word and see if it sounds all right. If not, add one of the vowels back. This will usually work out.

Instead of trying to explain why a given abbreviation has been chosen, we will just list the current list of abbreviations. If you have any questions about what the device abbreviation actually refers to please contact G. Trahern.

The device mnemonic is always used in the *DeviceName* part of the **SiteWideName.**

Mnemonic	Description
1int	interconnect # 1
2int	interconnect # 2
3int	interconnect # 3
4int	interconnect # 4
5int	interconnect # 5
6int	interconnect # 6
7int	interconnect # 7
b	beam position monitor (both planes)
bcb	big coil bottom (atr dipoles)
bct	big coil top (atr dipoles)
bd	beam dump
bh	horizontal beam position monitor
bs	beam stop
bv	vertical beam position monitor
c	collimator
cc	cold cathode gauge
cm	combined multipole magnet
cmn	combined multipole magnet (normal, b_n type)
cms	combined multipole magnet (skew, a_n type)
cq	quadrupole package with no sextupole

cqs	cqs corrector package
cqt	cqt corrector package
crs	crs corrector package
crt	crt corrector package
d	dipole
dec	decapole
decd	decapole
decf	decapole
dh	horizontal bending dipole
dm	For RHIC, magnetic component of dipole slot
dv	vertical bending dipole
f	flag monitor
foil	foil
fv	fast valve
fvs	fast valve sensor
ip	ion pump
jb	junction box
ka	dipole kicker abort
ki	dipole kicker injection
lamb	lambertson
lm	beam loss monitor
mark	marker
oct	octupole
octd	octupole
octf	octupole
p	verical pitching magnet
q	quadrupole
qd	defocusing quadrupole (horizontal plane)
qf	focusing quadrupole (horizontal plane)
qgt	gamma t quadrupole
qs	skew quadrupole
rv	roughing valve
scb	small coil bottom (atr dipoles)
sct	small coil top (atr dipoles)

skq	skew quadrupole
sv	sector valve
swm	switching magnet (between X and Y lines of ATR)
sx	sextupole
sxd	sextupole
sxf	sextupole
t	closed orbit corrector (both planes)
tb	terminal block
tc	thermocouple gauge
th	horizontal closed orbit corrector
tq	trim quadrupole
trp	trp corrector package
ttapb	thermal tap bottom (atr dipoles)
ttapt	thermal tap top (atr dipoles)
tv	vertical closed orbit corrector
vv	vent valve
xf	transformer

Bibliography

[1] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design*, Englewood Cliffs, New Jersey, Prentice Hall, 1991.

[2] *Select OMT*, Select Software Tools, 1526 Brookhollow Dr., Suite 84, Santa Ana, Ca 92706.

[3] V.M. Markowitz and A. Shoshani, "An Overview of the Lawrence Berkeley Laboratory Extended Entity-Relationship Database Tools", LBL Technical Report LBL-34932, 1993.

[4] T. DeMarco, *Structured analysis and system specification*, Englewood Cliffs, New Jersey, Prentice Hall, 1979.

[5] C. Saltmarsh, "ISTK Overview and the SDS Document", **RHIC AP** Note 29.

[6] LAMBDA reference manual, located in /rap/lambda/docs/NewDocs.

[7] C. Saltmarsh, "The *Glish* 2.4 User's Manual", **RHIC AP** note 30.

[8] ADO reference document, accessible via gopher in "some RHIC Documents Unmatched".

[9] ADO Interface Class Description and API, accessible via Mosaic with the url as http://acnsun10/RHIC/Controls/Documentation/AdoIf.fm.html

[10] ADOGEN and .rad files, accessible via Mosaic with the url as http://acnsun10/RHIC/Controls/Documentation/adogen.html

[11] Chapter 5 in this document and T. Satogata, **RHIC AP** Note 24.

[12] T. Satogata, **RHIC AP** Note 33.

[13] RHIC Design Manual, Chapter 10, H. Hahn et. al., editors.

[14] H. Grote and F. C. Iselin, "The MAD Program User's Reference Manual", CERN/SL/90-13 (AP) (Rev. 4)

[15] RHIC Design Manual, H. Hahn et. al., editors.

[16] Mike Harrison, RHIC Injection Beams Tests – Scope, memo dated May 18, 1994.

[17] Waldo MacKay, ATR talk for October 1994 MAC, RHIC/AP/39.

[18] Ted D'Ottavio, "SQL Front-End Comparison: Omnis 7 vs. PowerBuilder", AD/RHIC/RD-64 internal note, December 1994.