RHIC/AP/13
October 1, 1993

LAMBDA MANUAL

Waldo MacKay, Steve Peggs*,
Chris Saltmarsh, Todd Satogata
Brookhaven National Laboratory

Fady Harfoush, Jim Holt*, Leo Michelotti,
Francois Ostiguy, Al Russell
Fermi National Accelerator Laboratory

Eric Barr, Haregu Ferede,
Selcuk Saritepe, Garry Trahern*
Superconducting Super Collider

*Names marked with asterisks are primary contacts at their institutions.

Contents

The LAMBDA Collaboration 1

0. INTRODUCTION 2

0.1 Preamble 2

0.2 Environment Variables and Path 3

1. USERS GUIDE 5

1.1 SYBASE Interface for Lattice Users 5

1.1.1 dbsf Usage 5

1.2 SYBASE Interface for Lattice Designers 6

1.2.1 copy.in, copy.out - copy lattice files in or out of a lattice database 6

1.2.2 strength.in, strength.out - copy strength tables in or out of a lattice database 7

1.2.3 Lattice Table Formats 8

1.2.4 dwb - The Data WorkBench 9

1.2.5 dbsf Details 10

1.3 Survey 11

1.4 Twiss 13

1.4.1 lattice_function names 15

1.4.2 Structure of the SDS output file from twiss 16

1.5 Match 17

1.6 Graphical User Interface (GUI) 17

1.7 Bus, Support_Points Tables and the Wireup Program 19

2. DEVELOPERS GUIDE 21

2.1 Directory Structure 21

2.2 Flat format interface - flatin 22

2.3 BeamLine class interface - reader 23

2.4 flatin data structures 23

2.5 Survey data structures 29

2.5.1 Twiss data structures 33

2.5.2 BeamLine data structures 36

3. SYBASE SYSTEM ADMINISTRATORS GUIDE 37

3.1 Relational Databases and SQL for Lattice Description 37

3.2 Creating a lattice database 49

3.3 isql scripts 49

4. REFERENCES 51

5. APPENDIX 52

The LAMBDA Collaboration

The purpose of the LAMBDA collaboration is to develop a set of numerical models of accelerator performance, in a manner that allows simultaneous independent development by different authors at more than one site. Therefore, LAMBDA consists of several code modules that share common data structures, but that are compiled and run separately. In this sense the modules are 'Loosely Associated'. The collaboration makes its source code publicly available at no charge, and is open to the contribution of compatible modules by code authors anywhere.

The following list is limited to "active" LAMBDA collaborators - past or present code authors who will respond to e-mail questions about bugs or features in code modules. However, the collaboration is very grateful to many other accelerator physicists and computer scientists, not on the list, who have also contributed ideas and time to the project. They know who they are.

Please address any questions concerning the manual, or comments about errors and omissions, to Steve Peggs, peggs@owl.rhic.bnl.gov .

Brookhaven National Laboratory (BNL)

Waldo MacKay	mackay@owl.rhic.bnl.gov
Steve Peggs*	peggs@owl.rhic.bnl.gov
Chris Saltmarsh	salty@owl.rhic.bnl.gov
Todd Satogata	satogata@owl.rhic.bnl.gov

Fermi National Accelerator Laboratory (Fermilab)

Fady Harfoush	harfoush@calvin.fnal.gov
Jim Holt*	holt@calvin.fnal.gov
Leo Michelotti	michelot@calvin.fnal.gov
Francois Ostiguy	ostiguy@calvin.fnal.gov
Al Russell	russell@calvin.fnal.gov

Superconducting Super Collider (SSC)

Eric Barr	barr@fremont.ssc.gov
Haregu Ferede	haregu@fremont.ssc.gov
Selcuk Saritepe	saritepe@fremont.ssc.gov
Garry Trahern*	trahern@fremont.ssc.gov

*Names marked with asterisks are primary contacts at their institutions.

0. INTRODUCTION

0.1. Preamble

Traditional accelerator codes read the optical structure of an accelerator from an ASCII lattice file that has been written by a lattice designer. There are two significant disadvantages to this arrangement.

First, there are many different lattice file formats. It is practically impossible to maintain a consistent and accurate description of a given accelerator, even if it is only of modest complexity, when the description exists in.several parallel lattice files. One solution to this dilemma has been the voluntary adoption of a Standard Input Format (SIF) by many codes, including COMFORT, DIMAD, MAD, TEAPOT, TRANSPORT, and more[A]. This has not been completely successful, since there are many useful codes that do not conform to the SIF convention, and since there are minor SIF variants that lead to incompatibilities even between compliant codes. Another solution is to expand the functionality of a single code to satisfy a broad spectrum of user needs. This tends to result in large unwieldy codes that are inaccessible to the independent code developer.

The second disadvantage is that the ASCII interface is a compromise between the highly structured view of the lattice designer, and the flat nature of data input and storage. The lattice designer sees the accelerator heirarchically - (the machine has 6 sextants; sextant A has 23 cells followed by one low beta insertion; a cell has an F and a D half cell; an F half cell has an F quadrupole followed by 2 dipoles; et cetera). However, the computer stores the lattice information in a flat list (F quad, dipole, dipole, D quad, dipole, dipole, ...) and prefers to read it that way. At one end of the spectrum of compromises the code input routines are relatively simple, but the lattice designer is frustrated by having to edit a flat file containing much repetition of (perhaps) thousands of elements. At the other end of the spectrum, the lattice designer is satisfied, but the input routines that parse the complex syntax are long and complicated.

LAMBDA attempts to sever this Gordian knot by separating the files that the lattice designer edits from the ASCII or binary files that the accelerator code reads. A wide variety of lattice input formats may be generated from a single central description of a particular accelerator by using the utility **dbsf**. These formats include not only the most popular (highly structured) formats used by existing lattice codes, but also a very simple "flat" format. Independent code developers adopting the flat format can access existing accelerator descriptions with a minimum investment in lattice input routines. Intermediate

data storage - between designer ASCII files and lattice input files - is in the form of tables in a database in a Relational Database Management System (RDBMS). Thus, there are three different kinds of LAMBDA user - the *lattice designer/user*, the *code developer*, and the *database system administrator*. This manual is divided into three sections, addressing to their different needs.

A *lattice user* needs to know how to generate a lattice input file in a particular format by invoking the utility **dbsf** to translate from a set of database tables to that format. For example, in order to get a Proton Therapy Accelerator (pta) lattice file **pta.standard** ready for input to MAD, the user types **dbsf pta > pta.standard**. (The pta accelerator and database is used for example and demonstration purposes throughout this manual.) A *lattice designer* also needs to know how to generate the database tables that are in a one-to-one relationship with the ASCII files that he or she writes or modifies. This procedure is highly automated, requiring knowledge of two simple commands **copy.in** and **copy.out** described below (and possibly two other commands, **strength.in** and **strength.out**). No knowledge of the RDBMS is required.

Only the *database system administrator* needs to understand how to manipulate the RDBMS package itself, in order to activate database accounts, construct empty databases, et cetera. LAMBDA currently uses SYBASE to store the lattice database tables. SYBASE has been selected for site wide general use at BNL, for the next generation control system device database at Fermilab, and for technical accelerator specification at the SSC. A number of SYBASE *isql* scripts have been written, so that even a SYBASE novice can perform the usual tasks that are required to properly maintain lattice databases.

0.2 Environment Variables and Paths

Five environment variables need to be defined, and two path directories need to be added, in order to have full LAMBDA functionality. At Fermilab, for example, the user should add the following lines to his or her **.cshrc** file

```
setenv    LAMBDA        /usr/local/lambda
setenv    DBSF_DB       pta
setenv    DSQUERY       CALVIN
setenv    SYBASE        /usr/sybase
setenv    SYBASE_HOST   hobbes

set path = ($path $LAMBDA/bin $SYBASE/bin)
```

Of course, the environment variables need different definitions at different sites. The following table shows the appropriate definitions at the time that this manual was written.

Name	BNL	Fermilab	SSC
LAMBDA		/usr/local/lambda	/usr/local/lambda
DBSF_DB	—	pta	—
DSQUERY		CALVIN	SSCMOP
SYBASE		/usr/sybase	/usr/sakura/sybase
SYBASE_HOST		hobbes	sakura

The directories $LAMBDA/docs, $LAMBDA/demos and $LAMBDA/lattices/pta are open to the public and contain files of general interest.

1. USERS GUIDE

1.1 SYBASE Interface for Lattice Users

1.1.1 dbsf Usage

It is assumed that the user has set up appropriate environment variables, as described above. For example, on the accelerator subnets at Brookhaven and SSC, the SYBASE server is available over the network from any workstaion. So once the SYBASE and DSQUERY environment variables are defined, **dbsf** should work properly. As usual in the UNIX environment, typing **dbsf** results in a response describing the expected command line syntax, as follows

dbsf: mandatory key option missing
usage: dbsf [-CSMs8egfFbtp] [-V view_flags] [-T str_tbl_name] key [source_db]

-C	output an SDS file
-d	add dimension info to SDS file
-S	output a SYNCH format file
-M	output a MAGIC format file
-s	conform to MAD 4.03 standards
-8	conform to MAD 8.1 standards
-e	evaluate expressions
-g	group parameter, element and beam definitions separately
-f	produce old flat file format
-F	produce new flat file format
-b	try to produce gradient & brho vals for elements in SYNCH file this option must be run in conjunction with -S
-t	output a TRACE3D format file
-p	output a TEAPOT format file
-V view_flags	character flags indicating which view rules to apply
-T str_tbl_name	run with the specified strength table
key	the name of the item to be defined
source_db	db to search (default is environment variable DBSF_DB)

For example, the trivial query **dbsf pi** results in the following response for almost any lattice database

TITLE
! pta pi 08/24/92 14:15
pi := 3.14159265358979

This trivially conforms to SIF syntax. A more interesting response comes by saying

dbsf -ge pta pta > pta.standard or dbsf -F pta pta > pta.flat.
If the environment variable DBSF_DB for the source_db has been set to **pta**, the second of these commands reduces to **dbsf -F pta > pta.flat.** For more information, type **man dbsf.**

1.2 SYBASE interface for lattice designers

You must be logged on to the SYBASE_HOST to run the following commands. For example, at Fermilab type **rlogin hobbes.** It is also assumed that appropriate environment variables have been set, as described above.

1.2.1 copy.in, copy.out - copy lattice files in or out of a lattice database

copy.in and **copy.out** are executable scripts in the $LAMBDA/bin area, which is included in the users path if the appropriate environment variables have been set, as described above. The appropriate syntax is

Usage: copy.out <table_name>
Copies the table <table_name> out from the database specified by the environment variable DBSF_DB, to a file DBSF_DB.table_name. If <table_name> = 'all', all tables are copied out.

Usage: copy.in <table_name> <sybase_password>
[sybase_user_name]
Copies in to the table <table_name> in the database specified by the environment variable DBSF_DB, from the file DBSF_DB.table_name. If <table_name> = 'all', all tables are copied in.

as can be confirmed by typing simply **copy.in** or **copy.out** . For example, with the DBSF_DB environment variable set to "pta", try typing **copy.out all.** You do not need to have a SYBASE login to use **copy.out.** File names are automatically prepended by the name of the database. For example, typing **copy.out strength** generates the file "pta.strength" from the "pta" database.

In order to use **copy.in**, you must have appropriate permissions on the database, since database tables are overwritten. This means that you must have access to SYBASE as the database owner, or as the system administrator. By default, your SYBASE login name is assumed to be the same as your UNIX login name. The **copy.in** script automatically rebuilds the name_location table, so that subsequent calls to **dbsf** will work - unless there

is a logical or syntactical flaw in the files that have been read in. (See below for a discussion of file and table formats). File names are expected to be prepended by the name of the database. For example, the command **copy.in geometry password** will fail to refill the table "geometry" in the "pta" database, unless there is a file named "pta.geometry" in the local directory.

Uploading and downloading assumes that "bulk copying" has been enabled for the lattice database by the Sybase system administrator or database owner, when the (empty) database was created. It also assumes that the default user, "lattice_reader", has access to the database. In principle this is all transparent to the lattice designer.

1.2.2 strength.in, strength.out - copy strength tables in or out of a lattice database

A single database may have many different strength tables associated with it. For example, the Tevatron goes through as many as 17 "steps" during the "low beta squeeze" from injection optics to collision optics (see $LAMBDA/lattices/tev_92). Consequently, it is necessary to be able to manipulate files with the format of strength tables, but with arbitrary file name extensions after the database prepend. For example, the strengths for step 12 of the Tevatron low beta squeeze are loaded into the "tev_92" database by saying **strength.in step_12 password**, assuming that a file "tev_92.step_12" is present, and that DBSF_DB has been set to "tev_92".

Legal syntax for the scripts, which are also in $LAMBDA/bin, is essentially the same as for **copy.in** and **copy.out**. Typing **strength.in** or **strength.out** produces

Usage: strength.in <table_name> <sybase_password>
[sybase_user_name]
Copies in to the strength table <table_name> in the database specified by the environment variable DBSF_DB, from the file DBSF_DB.table_name. The strength table is either overwritten or, if necessary, created. You MUST be the database owner to create a table.

and Usage: strength.out <table_name> Copies the strength table <table_name> out from the database specified by the environment variable DBSF_DB, to a file DBSF_DB.table_name .

1.2.3 Lattice table formats

The ASCII files that **copy.in** (or **strength.in**) operates on must be properly formatted in order for the newly constructed database to perform properly. This is the responsibility of the lattice designer. A "hands on" way to learn the structure of the lattice tables, and the way in which they interact is to study how the lattice design files in the directory $LAMBDA/lattices/pta relate to the "pta" lattice input files by applying **dbsf** and **copy.out** appropriately. See the files in $LAMBDA/lattices/tev_92 for a relatively complex accelerator. For a pedagogical and complete description the legal element types, coordinate conventions, etcetera that **dbsf** conforms to, see the manual for MAD, version 8.03, and (for linac style elements) see the TRACE3D manual. For a terse description, see the section "flatin data structures", below.

There is a one-to-one correspondence between ASCII files and database tables. The tables consist of several rows with a fixed number of columns. Usually the first column is required to be filled with a unique "key" entry that is alphabetically sorted inside the database. Therefore, saying **copy.in all password**, followed by **copy.out all** results in alphabetized ASCII files. One row in a table corresponds to one line in an ASCII file, terminated by a newline (carriage return). Columns in the ASCII files are separated by tabs. Even if columns in a particular row are NULL (empty), the correct number of carriage returns must be present. The two most common causes for **copy.in** or **strength.in** to fail syntactically (with "bcp" errors reported to the screen) are the wrong number of tabs on a row, and a superfluity or a deficit of newlines.

Formal definitions of the table (and hence file) structures may be found in the directory $LAMBDA/sybase/formats . For example, the contents of the file "geometry.fmt" are

```
4.0
4
1       SYBCHAR0        20      ""      ""      1       name
2       SYBCHAR0        15      ""      ""      2       value
3       SYBCHAR0        90      ""      ""      3       definition
4       SYBCHAR0        130     "\n"    4       comment
```

In this case the first column contains the "name" of a geometry variable, with a maximum length of 20 characters, et cetera. The file $LAMBDA/lattices/pta/pta.geometry, reported verbatim, is

```
banglamb    -0.177923       Horizontal Lambertson bend angle
bangle      pi / 8.0        Bend angle of a dipole (m)
l1          0.36            Length of quad 1 in nozzle
```

l2	0.72	Length of quad 2 in nozzle
l21	0.0	Length from quad 2 to 1 in nozzle
l3	0.36	Length of quad 3 in nozzle
l32	0.0	Length from quad 3 to 2 in nozzle
l4	0.2	Length of quad 4 in nozzle
l43	1.4	Length from quad 4 to 3 in nozzle
lbpm	0.0	Length reserved for a Beam Position Monitor (m)
ldcorr	0.1	Magnetic length of a dipole corrector (m)
ldiparc	0.88	Magnetic arc length of a dipole (m)
dlamb	ldiparc−llamb	Length of drift complentary to Lambertson (m)
free	0.28	Length of free space for correctors, etcetera
lhalf	ldiparc+lquad+lfree	Length of a regular half cell (m)
lhalfl	llong+lquad+lfree	Length of a long half cell (m)
llamb	0.4	Magnetic length of horizontally bending Lambertson (m)
llong	2.18	Length of empty drift in long cells
lpatient	1.2	Length from nozzle quad 1 to patient
lquad	0.14	Magnetic length of a quadrupole (m)
lquart	0.5*(lhalfl−lquad−14)	Length of drift before quad 4in nozzle
lsext	0.1	Magnetic length of a sextupole (m)
lx1	1.386	Length of first drift in lengthener (m)
lx2	1.227	Length of second drift in lengthener (m)
pi	3.1415926535897931	Well known geometric constant

Note that the columns do not line up, since the column entries have variable length.

1.2.4 dwb - the Data WorkBench

Direct access to a lattice database in SYBASE is possible through the proprietary **dwb** X windows menu driven interface so long as the user has a SYBASE login and appropriate permissions on the database in question. It is possible to completely avoid use of the **copy.in**, et cetera, scripts, by using **dwb** and its various facilities. However, most lattice designers use **dwb** only rarely (if at all) as a powerful debugging tool that allows access to the guts of the database.

The use of **dwb** is described in great detail in the SYBASE manual "Data Workbench Users Guide", although it is possible to understand much of its functionality without referring to the manual. In order to use **dwb**, the user must be logged on to SYBASE_HOST. For example, a user at Fermilab, sitting at a workstation "garfield", must say

rlogin hobbes
cd appropriate_directory
setenv DISPLAY garfield:0.0
dwb

The third line here is commonly required to run X windows across a network. This fails if "garfield" is not open to displays generated by "hobbes". In this case type **xhost+** on "garfield", a command which may also be included in the users ".xinitrc" file.

1.2.5 dbsf Details

This section is included to illustrate and explain some of the less intuitive data base to output format that **dbsf** implements. (The reader may want to peruse the SYBASE System Administrator section before continuining.)

multipoles are included in the data base with their KnL and Tn multipole moments. Blank or NULL fields are assumed to be zero. A '\' in one of the Tn locations is meant to designate a skew moment with angle PI/(2n+2). A '\' in any field other than a Tn field is reported as an error by the various output routines. Standard input format can specify multipoles easily with skew moments represented by Tn values without numerical assignments. That is, "mmm: multipole, k1l = 1, k2l = 2, t1, t2 = .1" where t1 is a skew moment. SYNCH however can only designate multipoles with single multipole moments and skew or 0 tilt values. This is overcome by declaring an NPOL for each moment with the skew bit set if it is a skew, or alternately a ROT for arbitrary moment rotations. The flat format prints the KnL and Tn values as a list of evaluated values. If the moment is skewed, the value of PI/(2n+2) is evaluated and printed. MAGIC and Trace3d ignore multipoles.

kicker elements are kicks in both planes. This is trivial in SIF as it is an element type. However, SYNCH kicks are zero length, single plane elements. To solve this, a drift space of length orig_length/2 is declared, followed by a kick for each plane and another drift space. These are combined into a beam line which is named after the original element.

extended data tables have been added to accommodate elements with more attributes than will fit in the magnet_piece table of the database. The second function of these tables is to prevent the overloading of the strength column of the magnet_piece table. Currently, it is used to hold angles, multipole moments, kicks, frequencies and electrostatic field attributes. In any case, if data is found in the extended tables for a given element, it is used in place of the info in the magnet_piece table. In the case of multipoles and kickers, additional info must be included or an error message is printed.

1.3 Survey

The correct syntax for using the **survey** command is found by typing **survey** with no arguments, with results as follows.

Usage: survey [-fhu] [-T transverse_size_file_name] flat_file_name

'flat_file_name' contains lattice information in 'flat' format
'transverse_size_file_name' contains transverse size information Default units are meters
-f for output in international feet [1 inch = 2.54 cm]
-u for output in US survey feet [39.37 inches = 1 m]
-h turns off file headers
-T transverse_size_file_name to read transverse sizes of elements

'survey' also expects to find a 'survey.cmd' file containing commands

At least two input files must be present for **survey** to function properly. The first is "flat_file_name", a lattice file describing a beam line in flat format, generated by saying, for example, **dbsf -F beam_line > flat_file_name**. The second is a control file, "survey.cmd", which contains commands in a syntax described below. A third file describing the shape and transverse size of magnetic elements of the beam line may also be input. This file conforms to the format found in $LAMBDA/sybase/formats/magnet_size.fmt . See $LAMBDA/lattices/pta/pta.magnet_size for a working example.

As many as five files may be output from **survey**, including three files in TOPDRAWER graphics format, if "survey.cmd" file contains the appropriate commands. A file called "survey.log" is always output, to give the user a way to trace the operation of **survey** should something go wrong. If the command keyword "survey" appears in "survey.cmd", then survey coordinates and angles are sent to the standard output. If the command keyword "topdrawer" appears, then three files in TOPDRAWER graphics format are generated -"plan.top", "side.top", and "end.top", representing three projections of the beam line.

Figures 1, 2, and 3 show plots from three such graphics files that may be found in the directory $LAMBDA/demos area. They would be produced, for example, by saying **survey -T pta.magnet_size pta.flat**, when "survey.cmd" contained the following three active lines, as well as several comments

! comment line following an exclamation mark
survey

topdrawer !comment following an exclamation mark
stop
several more lines of comments and notes

Legal syntax in the "survey.cmd" file is demonstrated in the file $LAMBDA/demos/survey.cmd. In the following formal descriptions of allowable syntax, square brackets [ ] enclose optional arguments, and a pound sign # represents an alphanumeric number. The commands are case sensitive.

origin [x #] [y #] [z #] [fi #] [si #] [# theta] [s #]

Move the geometrical location of the beginning of the beam line to the specified location. Coordinate conventions are exactly as described in the MAD manual, version 8.03 . Moving the origin is useful when working in absolute coordinates, for example, relating several beam lines to each other in laboratory site coordinates. It is also useful in making isometric graphic projections that are often easier to visualize than the three simple projections of a beam line. For example, Figure 4 is an isometric view of the PTA accelerator corresponding to Figures 1 through 3.

print [element_name] ... [element_name] [all] [#s/e]
Print survey coordinates in the standard output file only at the named magnet elements (and the beginning and end of the beam line). Including "all" or "#s/e" in the list of names guarantees that coordinates will be printed at all elements.

stop
No more of the "survey.cmd" file is read beyond this statement, and **survey** exits successfully.

survey
Coordinates are calculated, stored, and reported to the standard output after this command has been given.

topdrawer
Create the three TOPDRAWER graphics files "plan.top", "side.top", and "end.top". This command must be preceded by a "survey" command. To find out more about how to handle TOPDRAWER graphics locally - for example, to make a landscape postscript file from a portrait TOPDRAWER file - see the file $LAMDA/docs/topdrawer.README .

Anything after **stop** is ignored. Anything after an exclamation point **!** anywhere in a line is a comment. Blank lines are skipped. The following characters {// (a space), \t (tab),

&, =, %} are interpreted as breaks between command line arguments . Command lines may not include a newline (\n, carriage return). For more information, see the section below titled "survey data structures".

1.4 Twiss

The program **twiss** calculates lattice functions (Twiss functions, etc.) from a lattice description which is either in the **FLAT** or **SDS** formats. Two input files must be present for twiss to function properly. The first is an input file which specifies the lattice either in **FLAT** or **SDS** format. The *lattice_file* file may be generated by the command:

dbsf *-switch beam_line* > *lattice_file*

where *switch* is either **-F** for the **FLAT** format or **-C** for the **SDS** format. The second is a control file, **twiss.cmd** , which contains commands in a syntax described below. A third file may be specified inside the **twiss.cmd** structure, if the input Twiss parameters to a beam line are the periodic conditions of a subsidiary line.

The correct syntax for running **twiss** is found by typing **twiss** with no arguments, which results in the following message:

```
Usage:  twiss [-ghsv] flat_file_name
  flat_file_name is a beam line in 'flat' or 'SDS' format
  -g to format standard output like 'tev_config'
  -h to NOT send headers to standard output
  -s to use shared memory rather than disk files
  -v for verbose logging output to 'twiss.log'
  twiss also expects to find a 'twiss.cmd' file
```

For example, typing

setenv DBSF_DB pta
dbsf -C pta
twiss -s pta.pta

would calculate lattice functions for the **pta** beam line from the **pta** database using an **SDS** format input file.

In addition to *stdout*, as many as four files may be output from twiss, including a file in TOPDRAWER graphics format, if the **twiss.cmd** file contains the appropriate commands. A file called *twiss.log* is always output, to give the user a way to trace the operation of twiss should something go wrong. If the command keyword **twiss** appears in *twiss.cmd*, then Twiss functions are sent to the standard output. If the command keyword **topdrawer** appears, then the TOPDRAWER graphics files *twiss.top*, *side.top*, and *end.top* are also

generated. An **SDS** file *Twiss* is written with the values of the lattice functions at the end of each element in the beamline. The **-s** switch causes **twiss** to use shared memory rather than disk files for the lattice description input file and *Twiss* output file.

The **twiss.cmd** file consists of a list of action commands with arguments and comments. A comment is considered to be anything on a line after an exclaimation point (!).

In the following formal descriptions of allowable syntax, square brackets ([ ]) enclose optional arguments. An ellipsis (...) is used to indicate repeated arguments. The command interpereter is case sensitive, so the commands should be typed in lower case as shown. An *italic font* is used to indicate a choice of values. Specific file names are indicated by a *bold slanted font.*

Here is a description of the possible commands, followed by the possible choices for some of the command parameters.

1. **deltap** *value*

   Set the off-momentum parameter delta_p/p_nominal to a desired value.

2. **gammat_sens**

3. **initial_offset**

   where *offset_par* is one of the following offsets:

s_offset:	s-offset
mu_x_offset:	μ_x-offset
mu_y_offset:	μ_y-offset

4. **initial** [*lattice_function value* [,...]]

   Set the initial Twiss parameters and beam envelope parameters at the beginning of the main beam line to be these values. Another way to set them is to use the periodic command (see below). Default values are **beta_x** = **beta_y** = 1.0, all others zero. Acceptable entries for *lattice_function* are given in Section 1.4.1.

5. **periodic** [*second_flat_filename*]

   Signifies that the initial Twiss parameters are to be taken from the periodic solution of a beam line. If no additional argument is given, the periodic solution for the main beam line is used, but if a second valid flat file name is given, then its periodic solution will be used.

6. **s_offset** *value*

   Adjust the initial azimuth of the beam line to the specified value.

7. **twiss**

   Twiss parameters are calculated, stored, and reported to the standard output after this command has been given.

8. **football** [*football_par*] [,...]

   The **football** command propagates the beam envelope functions. It is similar to the **twiss** command, except that it accepts arguments which specify the various functions to be tabulated in the *stdout* output file. If no *football_par*'s are specified, it prints the standard ten functions: β_x, α_x, β_y, α_y, η_x, η'_x, $\mu_x/(2\pi)$, and $\mu_y/(2\pi)$. The acceptable entries for *football_par* are discussed in Section 1.4.2.

9. **topdrawer**

   Create the TOPDRAWER graphics files *twiss.top*. This command must be preceded by the command **twiss** . To find out more about how to handle TOPDRAWER graphics locally – for example, to make a landscape postscript file from a portrait TOPDRAWER file. (See *$LAMDA/docs/topdrawer.README.*)

10. **stop**

    No more of the file **twiss.cmd** is read beyond this statement, and **twiss** exits successfully.

11. **print** [*element_name*]
    print all
    print *start/end*

    Print survey coordinates in the standard output file only at the named magnet elements (and the beginning and end of the beam line). Including **all** or *start/end* in the list of names guarantees that coordinates will be printed at all elements.

Anything after **stop** is ignored. Anything after an exclaimation point (!) is ignored as a comment. Blank lines are skipped. White space (tabs and spaces), as well as ampersands (&), equals signs (=), and percent signs (%) are treated as breaks between arguments.

1.4.1 *lattice_function* names

Valid names for *lattice_function* are the usual uncoupled lattice functions (Twiss parameters and dispersion functions):

1. **beta_x:** $\beta_x(s)$
2. **beta_y:** $\beta_y(s)$
3. **alfa_x:** $\alpha_x(s)$
4. **alfa_y:** $\alpha_y(s)$
5. **eta_x:** $\eta_x(s)$
6. **eta_y:** $\eta_y(s)$
7. **eta_xp:** $\eta'_x(s)$
8. **eta_yp:** $\eta'_y(s)$

and the beam envelope correlation functions:

9. **xx:** $\langle x^2 \rangle$

10. **xxp:** $\langle xx' \rangle$
11. **xy:** $\langle xy \rangle$
12. **xyp:** $\langle xy' \rangle$
13. **xz:** $\langle xz \rangle$
14. **xu:** $\langle x\delta \rangle$
15. **xpxp:** $\left\langle (x')^2 \right\rangle$
16. **xpy:** $\langle x'y \rangle$
17. **xpyp:** $\langle x'y' \rangle$
18. **xpz:** $\langle x'z \rangle$
19. **xpu:** $\langle x\delta \rangle$
20. **yy:** $\langle y^2 \rangle$
21. **yyp:** $\langle yy' \rangle$
22. **yz:** $\langle yz \rangle$
23. **yu:** $\langle y\delta \rangle$
24. **ypyp:** $\left\langle (y')^2 \right\rangle$
25. **ypz:** $\langle y'z \rangle$
26. **ypu:** $\langle y'\delta \rangle$
27. **zz:** $\langle z^2 \rangle$
28. **zu:** $\langle z\delta \rangle$
29. **uu:** $\langle \delta^2 \rangle$

Valid names for *football_par* are the same as for *lattice_function* with the additional functions:

30. **gamma_x:** $\gamma_x(s)$
31. **gamma_y:** $\gamma_y(s)$
32. **eta_z:** (This is here for completeness, although it is not really η_z).
33. **eta_1:** $= 1$, unless there is a longitudinal kick.
34. **mu_x:** $\mu_x(s)$
35. **mu_y:** $\mu_y(s)$

1.4.2 Structure of the SDS output file from twiss

The **SDS** output file *Twiss* contains two c-structures of the following typedef's:

```
typedef struct { /* Recognized by SDS as ''Twiss_input_file'' */
char filename[40];
} flat_file;
typedef struct { /* Recognized by SDS as ''Twiss'' */   int
lattice_index;
  int element_index;
  int type_index;
  double pathlen;
  double beta_x;
  double alfa_x;
  double gamma_x;
  double beta_y;
```

```
    double alfa_y;
    double gamma_y;
    double eta_x;
    double eta_xp;
    double eta_y;
    double eta_yp;
    double eta_z;
    double eta_1;
    double mu_x;
    double mu_y;
    double bm_xx;
    double bm_xxp;
    double bm_xy;
    double bm_xyp;
    double bm_xz;
    double bm_xu;
    double bm_xpxp;
    double bm_xpy;
    double bm_xpyp;
    double bm_xpz;
    double bm_xpu;
    double bm_yy;
    double bm_yyp;
    double bm_yz;
    double bm_yu;
    double bm_ypyp;
    double bm_ypz;
    double bm_ypu;
    double bm_zz;
    double bm_zu;
    double bm_uu;
} optic[];
```

1.5 Match

1.6 Graphical User Interface (GUI)

The LAMBDA graphical user interface (GUI) makes a bare-bones, no-frills, minimal options set of primitive LAMBDA modules immediately available to a novice or infrequent user as painlessly as possible. With it, he can pull a lattice out of the database,write it to a file in a number of formats, and use the file as input to other LAMBDA modules. On invocation, the interface appears iconified, as shown in the upper right of g. Upon opening the icon, the user is presented with three primary buttons labelled "Lattice," "File format," and "LAMBDA modules." Briefly, the functions of these buttons are:

* Lattice

The menu and submenus attached to this button allow the choice of a lattice that has been installed in the database. When the desired lattice has been selected, its SYBASE name appears in the "Lattice" text field, under "Current settings." Rather than use this button or the others, the user can simply write that name into the text field (if he happens to know it). Entering a carriage return then selects it.

* LAMBDA modules

Modules are invoked by choice within the "LAMBDA modules" menu. If the user has not yet specified enough information to call a module, the GUI aborts the request and asks for the necessary input. The Lambda modules which currently exist are:

survey Produces a two-dimensional pictorial representation of the accelerator or beamline as it is geometrically laid out.

extract Uses the dbsf module to read a lattice from the database and write an ASCII (or SDS) lattice file in a chosen format.

twiss Computes and graphs the linear lattice functions.

As other modules come into existence they will be added to the menu.

* File format

Selecting "extract" produces a lattice file in one of a number of formats. The "File format" menu enables the user to select one of the four currently supported by dbsf: MAD, SYNCH, SDS, and FLAT.

Figure 1 caption.Clicking on the "About Lambda" text field produces a short message listing the authors of the various LAMBDA modules and how to contact them. The Lambda GUI was programmed assuming OPEN LOOK widgets. It will work with a Motif window manager, but it will not work on a DEC workstation since it is not compatible with Digital's mouse. The user must be logged on to a computer that has access to SYBASE and dbsf. At Fermilab this is HOBBES.

1.7 Bus, Support_Points Tables and the Wireup Program

As an optional structure we have provided the ability to describe the ganging of magnets on power supply buses in the lattice database. This structures uses two additional tables, bus and support_points, as well as the SDS output from DBSF. The program WIREUP uses both the lattice SDS and these two tables to create another SDS dataset which assigns to each element a bus name, a sequence number on that bus starting from the point where the power supply bus enters the lattice, as well as a current (I) orientation.

A complete description of how to use the WIREUP program is contained in the WIREUP manual page. Please refer to it for step by step usage and examples.

The SQL for the bus and support_point tables is

```
create table bus
(name                    char(20),
level                    char(1)                 null,
elements                 varchar(150)            null,
comment                  varchar(130)            null)
create unique clustered index bus_index on bus(name)
go

create table support_points
(name                    char(20),
type                     varchar(20),
system_name              varchar(20),
displacement             varchar(20)             null,
comment                  varchar(130)            null)
go


create rule level_rule
as @level in ('r','l','c','x','0')
go


sp_bindrule level_rule, "bus.level"
go


grant select on bus to public
grant select on support_points to public
go
```

A bus is a pattern of magnets with the current either flowing to the right or the left with respect to a given (say clockwise = right) direction through the lattice. The bus can be described just like a beam line, i.e., as a sequence of elements, but unlike a beam line the bus only includes the lattice elements which are on the specific bus. It it the job of WIREUP to associate a given bus with the lattice. By assigning to each sub pattern of a bus the level 'r' or 'l', one can build up any current configuration with the proviso that a given magnet is only on **one** bus.

To correlate the bus with the lattice **requires** the addition of markers to the lattice description by modifying some of the beam lines or slots. There are two kinds of support points that WIREUP understands. One is called 'masterps' and the other is 'turnaround'. In the magnet_piece table both of these bus support points are declared to be of type **marker**. The turnaround marker is located at the left most part of its bus in the lattice and the masterps marker is located at the point in the lattice where the power supply line enters the lattice. The numerical sequencing of magnets on the bus will initiate at the masterps location. These two markers need not be entered into the bus table. The relation between the support_points table and the bus table is established by the value of the **system_name** column of the support_points table. The system_name and bus name should correspond for the appropriate support points.

Finally, for WIREUP to function properly the **name** column from the bus table must be inserted into the name_location table. Because of the unique index on the name column of the name_location table, one must choose bus names that do not conflict with any other names used in the lattice description.

2. DEVELOPERS GUIDE

2.1 Directory Structure

$LAMBDA/**bin**	contains executables and scripts.
$LAMBDA/**demos**	contains example files, and is a good place to play around in.
$LAMBDA/**docs**	contains README files and useful documentation, including this manual.
$LAMBDA/**gui**	contains
$LAMBDA/**include**	contains all include files, include_file_name.h .
$LAMBDA/**lib**	contains liblambda.a, a library of object files. It should be unnecessary to link to object files in any other directory. Source code for the various modulesis found in the appropriate module directory, such as $LAMBDA/flatin, for example.
$LAMBDA/**sybase**	contains isql script files, and a subdirectory, /formats, that contains the formal definitions of the lattice database tanles/files. Obsolete scripts may be found in the /save subdirectory.

ADD DIRECTORIES AND COMMENTS

2.2 Flat format interface - flatin

The program DBSF can extract the database lattice information in two types of flat formats. One is a flat ASCII format and the other is a flat **SDS** format. The "flatin" function opens and reads the flat format file produced by dbsf, dynamically allocates memory, fills the lattice structures and returns a pointer table so that the application can access the data. The "flatin_sds" function does the same thing except the input file is an **SDS** file. Both functions have the same passing parameters:

status = flatin(char * file_name, flatin_data *my_data);

where the "file_name" is the name of the flat file and "my_data" is a pointer to the lattice structures pointer table.

A sample program using the flatin function call is:

```
/************* This demo demonstrates and tests flatin.c functions ******************/
#include "flatin.h"

main(argc, argv)
    int argc; char **argv; {
    flatin_data my_flatin_data;
    char *file_name;
    char latfile[NAME_MAX];

if ( argc != 2) {
    printf("Usage: flatin_demo flat_file_name0);
    printf("flat_file_name is a beam line in 'flat' format0); exit(1); } else {
    file_name = *(argv+1);

strcpy(latfile,file_name);
    }

if (flatin(latfile,&my_flatin_data)) {
    printf("found4-5d parameters,4-5d elements,",
    my_flatin_data.number_of_parameters,
    my_flatin_data.number_of_elements);
    printf("5-5d elements in the beamline0,
    my_flatin_data.number_of_lattices);

    }
```

```
exit(0);
}
 /*****************************************************************/
```

This program can found in $LAMBDA/flatin as flatin_demos.c

2.3 BeamLine class interface - reader

2.4 flatin data structures

There are five structures which are used to describe a lattice. They are **parameter, element, atom, legal_type,** and **row**.

The **parameter** structure as its name implies, holds the name and value of all of the parameters extracted for a particular lattice. Also, if an element has more than a length, strength and tilt, the additional values are stored in the parameter structure.

The **element** structure is the basic unit of the lattice. Each element has a name, a type name as defined in the **legal_type** structure, and possibly a length, strength and tilt. If length, strength or tilt is defined from a parameter then both the value and index into the parameter structure variables are filled in. If there are additional parameters beyond length, strength and tilt, then npars is nonzero and more_index points to the location in the **parameter** structure where the additional parameters are located. These parameters are placed consecutively.

An array of **atom** structures constitutes the lattice. In the structure is an index to the pertinent **element** structure and an index to the **element** type. The other variables are not used by flatin.

The **legal_type** structure is a pre-initialized structure which lists all of the available element types and the characteristics associated with that element.

The **row** structure is also a pre-initialized structure which contains a list of the additional parameters for the elements which have more than length, strength and tilt.

The following is a direct reproduction of the file $LAMBDA/include/flatin.h.

```
**********************************************************************

#ifndef FLATIN_H
   #define FLATIN_H

   #define LATTICE_BASE_LEVEL 0
```

```
#define LATTICE_SLOT_LEVEL 1
#define LATTICE_SUPERSLOT_LEVEL 2

/* These define the row_struct array without using can pointers: somewhat wasteful
of
space but will allow fairly easy
transport between programs and languages even without SDS
*/
#define NAME_MAX 64
#define COLUMN_MAX 22

/* These two are not needed if SDS is used. If SDS isn't used, they should be derived
so we presume lattice_inits.h has been loaded..... */

#ifndef SDS_MAGIC
#define SDS_MAGIC
#define NUMBER_OF_COLUMN_ENTRIES sizeof(row)/sizeof(row_struct) -1
#define LEGAL_TYPE_COUNT sizeof(legal_type)/sizeof(legal_struct) #endif

typedef struct {
char name[NAME_MAX];
double value;
} parameter_struct;

typedef struct {
char name[NAME_MAX];
char type[NAME_MAX];
char sub_type[NAME_MAX];
double length;
double strength;
double tilt;
int type_index;
int npars;
int length_index;
int strength_index;
int tilt_index;
int more_index;
} element_struct;
```

```
typedef struct {
double s;
int element_index;
int type_index;
int occurence;/* not filled by flatin */
int level;/* not filled by flatin */
int hook_index;/* not filled by flatin */
int sense;/* not filled by flatin */
} atom_struct;

typedef struct {
char name[NAME_MAX];
int has_length;
int has_strength;
int has_tilt;
int has_more;
} legal_struct;

typedef struct {
char name[NAME_MAX];
int cols;
char heading[COLUMN_MAX][NAME_MAX];
} row_struct;

typedef struct {
int morend;
double attribute[COLUMN_MAX];
} more_struct;

typedef struct {
parameter_struct *parameter_ptr;
int number_of_parameters;
element_struct *element_ptr;
int number_of_elements;
atom_struct *atom_ptr;
int number_of_atoms;
legal_struct *legal_type_ptr;
```

```
int number_of_legal_types;
row_struct *row_ptr;
int number_of_rows;
} flatin_data;

#endif /* FLATIN_H */
```

****

The pointer table of type **flatin_data** is the main interface with programs that call flatin.

The structures **legal_type** and row are filled with fixed values, read from the file $LAMBDA/include/lattice_inits.h that is reproduced below.

****

```
#ifndef LATTICE_INITS_H
#define LATTICE_INITS_H

legal_struct legal_type[] = {
"",                    0,0,0,0, /* NULL element */
"drift",               1,0,0,0, /* length */
"hmonitor",            1,0,0,0, /* length */
"vmonitor",            1,0,0,0, /* length */
"monitor",             1,0,0,0, /* length */
"instrument",          1,0,0,0, /* length */
"wiggler",             1,0,1,0, /* length, tilt */
"rbend",               1,1,1,1, /* length, angle, tilt, more */
"sbend",               1,1,1,1, /* length, angle, tilt, more */
"quadrupole",          1,1,1,0, /* length, k1, tilt */
"sextupole",           1,1,1,0, /* length, k2, tilt */
"octupole",            1,1,1,0, /* length, k3, tilt */
"multipole",           0,0,0,1, /* more */
"solenoid",            1,1,0,0, /* length, ks */
"rfcavity",            1,0,0,1, /* length, more */
"elseparator",         1,1,1,0, /* length, e, tilt */
"srot",                0,1,0,0, /* angle */
"yrot",                0,1,0,0, /* angle */
```

```
"hkick",          1,1,1,0, /* length, hkick, tilt */
"vkick",          1,1,1,0, /* length, vkick, tilt */
"kicker",         1,0,1,1, /* length, tilt, more */
"marker",         0,0,0,0,
"ecollimator",    1,0,0,1, /* length, more */
"rcollimator",    1,0,0,1, /* length, more */
"tbend",          1,1,1,1, /* length, angle, tilt, more */
"thinlens",       0,0,0,1, /* more */
"tank",           1,0,0,1, /* length, more */
"edge",           0,0,0,1, /* more */
"pmq",            1,0,0,1, /* length, more */
"rfqcell",        1,0,0,1, /* length, more */
"doublet",        1,0,0,1, /* length, more */
"triplet",        0,0,0,1, /* more */
"rfgap",          0,0,0,1, /* more */
"special",        1,0,0,0, /* length */
"rotation",       0,1,0,0, /* angle */
"beamline",       1,0,0,0, /* length */
"slot",           1,0,0,0, /* length */
"end_of_list",    0,0,0,0
};

row_struct row[] = {
    "",0,"","","","","","","","","",
    "","","","","","","","","","","","","","",
    "ecollimator", 2, "xsize", "ysize", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "",
    "",
    "kicker", 2, "hkick", "vkick", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "",
    "multipole", 20, "k0l", "k1l", "k2l", "k3l", "k4l", "k5l", "k6l", "k7l", "k8l", "k9l", "t0", "t1", "t2",
    "t3", "t4", "t5", "t6", "t7", "t8", "t9", "", "",
    "rbend", 9, "k1", "k2", "k3", "e1", "e2", "fint", "hgap", "h1", "h2", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "",
    "","","","","",
```

"rcollimator", 2, "xsize", "ysize", "", "", "", "", "", "", "", "", "", "", "", "", "", "",
"", "", "", "", "",
"",

"rfcavity", 7, "volt", "lag", "harmon", "betrf", "pg", "shunt", "tfill", "", "", "", "",
"", "", "", "",
"", "", "",
"", "", "", "",

"sbend", 9, "k1", "k2", "k3", "e1", "e2", "fint", "hgap", "h1", "h2", "", "", "", "",
"", "", "", "",
"", "", "", "",
"",

"tbend", 22, "b0", "a0", "b1", "a1", "b2", "a2", "b3", "a3", "b4", "a4", "b5", "a5",
"b6", "a6",
"b7", "a7", "b8", "a8", "b9", "a9", "b10", "a10",

"thinlens", 3, "xfocal", "yfocal", "zfocal", "", "", "", "", "", "", "", "", "", "", "", "",
"", "", "", "",
"", "",
"",

"doublet", 1, "distance_between", "", "", "", "", "", "", "", "", "", "", "", "", "", "",
"", "", "", "",
"", "",
"",

"triplet", 5, "strength_outer_quad", "length_outer_quad", "distance_between",
"strength_inner_quad", "length_inner_quad", "", "", "", "", "", "", "", "", "", "", "",
"", "", "", "",
"",
"",

"edge", 5, "rotation_angle", "radius", "gap", "fringe_field1", "fringe_field2","", "", "",
"", "", "", "", "", "", "", "", "", "", "", "", "",
"",

"rfqcell", 4, "V/(r**2)", "AV", "phase", "type", "", "", "", "", "", "", "", "", "", "",
"", "", "", "",
"", "", "",
"",

"rfgap", 5, "effective_gap_voltage", "phase", "emittance_growth_flag",

```
    "energy_gain_flag", "harmonic", "", "", "",
    "", "", "", "", "", "", "", "", "", "", "", "", "", "",
    "",
    "pmq", 2, "inner_radius", "outer_radius", "", "", "", "", "", "", "", "", "", "", "", "", "", "",
"", "", "", "",
    "", "", "", "",
    "",
    "tank", 3, "effective_accel_gradient", "inj/exit_phase", "number_of_iden_cavities",
    "", "", "", "", "",
    "", "", "", "", "", "", "", "", "", "", "", "", "", "",
    ""

    };

    #endif /* LATTICE_INITS_H */
```

****

2.5 Survey data structures

The following are direct reproductions of the files $LAMBDA/include/survey.h and $LAMBDA/include/survey_dict.h.

****

```
#ifndef SURVEY_H
#define SURVEY_H

#define TRUE                    1
#define FALSE                   0
#define ARC_SEGMENTS            4
#define BUFF_MAX                7                   128 *
#define HEADER_PERIOD           50
#define MAX_TOKEN_SIZE          132
#define MAX_TOKEN_COUNT         100
#define TOP_BUFF_MAX            1024
#define METERS                  0
#define INTERNATIONAL_FEET      1
#define US_FEET                 2
```

```
#define RBEND                   7
#define SBEND                   8

typedef struct {
    double *R;
    double **S;
    double **T;
    double int_width;
    double ext_width;
    double int_height;
    double ext_height;
    int repout;
    int sizeout;
    int nrep;
    } local_matrix_struct;

typedef struct {
    double X;
    double Y;
    double Z;
    double fi;
    double si;
    double theta;
    double **W;
    } coord_struct;

typedef struct {
    FILE *topx;
    FILE *topy;
    FILE *topz;
    double *temp;
    coord_struct *c_ptr;
    element_struct *e_ptr;
    local_matrix_struct *l_ptr;
    } top_struct;

typedef struct {
```

```
    time_t time_stamp;
    char flat_file_name[NAME_MAX];
    char size_file_name[NAME_MAX];
    double s_off;
    double xmin;
    double xmax;
    double ymin;
    double ymax;
    double zmin;
    double zmax;
    double height_max;
    double width_max;
    double **init_FI;
    double **init_SI;
    double **init_THETA;
    coord_struct *coord_ptr;
    flatin_data *flatin_ptr;
    local_matrix_struct *locmat_ptr;
    } survey_data;

/*-------------------------------------------------------*/
/*          external functions and objects               */
/*-------------------------------------------------------*/

extern double *dim1();
extern double **dim2();
extern legal_struct legal_type[];
extern double pi, twopi;

#endif /* SURVEY_H */
```

****

$LAMBDA /include/survey_dict.h

****

```
#ifndef SURVEY_DICT_H
    #define SURVEY_DICT_H

    #define DICT_SIZE 64
    #define MAX_TOKEN_SIZE 132
    #define MAX_TOKEN_COUNT 100

    typedef char token[MAX_TOKEN_SIZE];
    typedef struct {
    token piece[MAX_TOKEN_COUNT];
    } array_of_tokens;

    enum command_type {no_keyword,
    print,mv_origin,stop,survey,topdrawer,cmd_error;
    typedef struct {
    enum command_type      dict[DICT_SIZE];
    token                  word[DICT_SIZE];
    } command_struct;
    command_struct command = {
    {no_keyword, print, mv_origin, stop, survey, topdrawer, cmd_error }, {"",
    "print", "origin", "stop", "survey", "topdrawer" } };

    enum origin_type { x, y, z, fi, si, theta, s_off, origin_error }; typedef struct
    {
    enum origin_type    dict[DICT_SIZE];
    token               word[DICT_SIZE];
    } origin_struct;
    origin_struct origin = {
    {x, y, z, fi, si, theta, s_off, origin_error},
    {"x", "y", "z", "fi", "si", "theta", "s"}
    };

    enum mark_type {all, mark_named_element};
    typedef struct {
    enum mark_type      dict[DICT_SIZE];
    token               word[DICT_SIZE];
    } mark_struct;
    mark_struct mark = {
```

```
{all, all, mark_named_element},
{"all", "#s/e"}
};

#endif /* SURVEY_DICT_H */
```

****

2.5.1 Twiss data structures

The following are direct reproductions of the files $LAMBDA/include/twiss.h and $LAMBDA/include/twiss_dict.h .

****

```
#ifndef TWISS_H
  #define TWISS_H

  #define FALSE 0
  #define TRUE 1
  #define BUFF_MAX 128
  #define HEADER_PERIOD 50
  #define TOP_BUFF_MAX 1024

  #define RBEND 7
  #define SBEND 8
  #define QUADRUPOLE 9
  #define SEXTUPOLE 10

  typedef struct {
  double **matrix;
  int repout;
  } magmat_struct;

  typedef struct {
  double beta_x;
  double alfa_x;
  double gamma_x;
  double beta_y;
  double alfa_y;
```

```
double gamma_y;
double eta_x;
double eta_xp;
double one_x;
double eta_y;
double eta_yp;
double one_y;
double psi_x;
double psi_y;
} optic_struct;

typedef struct {
double co_x;
double co_xp;
double co_y;
double co_yp;
double co_dt;
double co_delta;
} clorb_struct;

typedef struct {
time_t time_stamp;
char flat_file_name[NAME_MAX];
double delta;
double s_offset;
double beta_x_max;
double beta_y_max;
double eta_x_min;
double eta_x_max;
double eta_y_min;
double eta_y_max;
double quad_strength_max;
clorb_struct init_clorb;
optic_struct init_optic;
double **transfer_matrix;
clorb_struct *clorb_ptr;
flatin_data *flatin_ptr;
```

```
    magmat_struct *magmat_ptr;
    optic_struct *optic_ptr;
    } twiss_data;

/*-----------------------------------------------------------*/

/*              externals                */
/*-----------------------------------------------------------*/

    extern double pi, twopi, twopinv;
    extern double *dim1();
    extern double **dim2();
    extern legal_struct legal_type[];

    #endif /* TWISS_H */

        ***************************************************************

        $LAMBDA/include/twiss_dict.h

        ***************************************************************

    #ifndef TWISS_DICT_H
    #define TWISS_DICT_H

    #define DICT_SIZE 64
    #define MAX_TOKEN_SIZE 132
    #define MAX_TOKEN_COUNT 100

/*-----------------------------------------------------------*/
/*              dictionaries             */
/*-------------- ------------------------------------------*/

    typedef char token[MAX_TOKEN_SIZE];
    typedef struct {
    token piece[MAX_TOKEN_COUNT];
    } array_of_tokens;

    enum command_type {no_keyword, deltap, init_condx, offset, periodic, print,
    stop, topdrawer, twiss, cmd_error};
```

```
typedef struct {
enum command_type    dict[DICT_SIZE];
token                word[DICT_SIZE];
} command_struct;
command_struct command = {
{no_keyword, deltap, init_condx, offset, periodic, print, stop, topdrawer,
twiss, cmd_error, {"", "delta", "initial", "s_offset", "periodic", "print", "stop", "top-
drawer",
"twiss" } };

enum twiss_type beta_x, beta_y, alfa_x, alfa_y, eta_x, eta_y, eta_xp, eta_yp,
twiss_error};
typedef struct {
enum twiss_type      dict[DICT_SIZE];
token                word[DICT_SIZE];
} twiss_struct;
twiss_struct twiss_names = {
{beta_x, beta_y, alfa_x, alfa_y, eta_x, eta_y, eta_xp, eta_yp,
twiss_error},
{"beta_x","beta_y","alfa_x","alfa_y","eta_x","eta_y","eta_xp","eta_yp"} };

enum mark_type {all, mark_named_element};
typedef struct {
enum mark_type       dict[DICT_SIZE];
token                word[DICT_SIZE];
} mark_struct;
mark_struct mark = {
{all, all, mark_named_element},
{"all", "#s/e"}
};

#endif/* TWISS_DICT_H */
```

****

2.5.2 BeamLine data structures

3. SYBASE SYSTEM ADMINISTRATORS GUIDE

Standard Query Language (SQL) is used inside an RDBMS to get information that a user desires from database tables. SQL is (almost) universally adopted by all RDBMS vendors, including SYBASE. Scripts conforming to the SQL syntax, and stored in an ASCII file, may be loaded into the Data Workbench, and even stored as part of the database itself. Interactive access to SYBASE database information is also possible from the usual computer environment by invoking "isql" scripts, which consist mainly of commands in SQL syntax. Useful examples of these may also be found in the $LAMBDA/sybase directory.

A third way to access SYBASE database information is to embed calls from the proprietary "DB-Library" in a piece of application code. The code **dbsf**, for example, uses DB-Library calls in order to parse the lattice database structure. RDBMS manufacturers, including SYBASE, are in the process of adopting the "Embedded SQL" standard. If and when it is possible to rewrite **dbsf** using such standard embedded calls, LAMBDA will cease to be tied to a single RDBMS product.

For more details, or to go beyond the following "cookbook" procedures, see the various SYBASE manuals.

According to convention, the login name of the SYBASE system administrator is "sa", and SQL scripts become "procedures" when they have been copied into the Data Workbench.

3.1 Relational Databases and SQL for Lattice Description

The basic data structure used in a relational database is that of a table. The table is made of a finite set of columns and an arbitrary number of rows. The data type of each column is fixed when the table is created. An application program such as DBSF (User's Guide, section 1.1.1) which uses these tables must understand the data structure of the table. In this section we list the database related code which DBSF knows how to interpret. (By the way, the program name is actually 'dbsf', but we will capitilize it here for emphasis.) This code, specific to relational databases, is known as Structured Query Languange or SQL. We will not give a primer on the SQL language here, but while it is not a difficult language to learn, it is a highly nontrivial matter to use SQL efficiently. We believe the SQL listed below is fairly straightforward if not efficient, but for those who wish to learn more about SQL, the book 'The Practical SQL Handbook', by Sandra L. Emerson, Marcy Darnovsky and Judith S. Bowman of Sybase, Inc. (1989, Addison-Wesley) is very helpful. All of the SQL discussed in this chapter is incorporated into scripts that can be found in the $LAMBDA/sybase directory.

To construct tables appropriate to describe an accelerator lattice, we have abstracted from the MAD 8.1 beam line notation and element types as well as the TRACE3D element definitons. Comments on each table definition where appropriate will follow the SQL 'create' statements. We have chosen to use MAD and TRACE3D as the primary standard for element types since experience has shown that the types used by these codes can be mapped to other accelerator codes by DBSF.

Some general comments on the structure of the database are needful. There are five basic tables used to describe a lattice: geometry, strength, magnet_piece, beam_line and slot. Two utility tables, name_location and name_alias, are required by DBSF. In addition, there are 23 secondary tables which can be used to store all the additional parameters needed to define lattice elements a la MAD and TRACE3D.

In general, all columns in every table in the database have character data type. If the user enters a number (e.g. 3.4e-3) in a certain column, it is interpreted as a string by the database. However, when DBSF extracts information from the database, such strings are interpreted correctly as numerical data. (Note: all calculations with real numbers in DBSF use double precision.)

Finally, the units used in the database tables are those appropriate to MAD and TRACE3D. For tables used within the MAD schema, MKSA units are accepted, and for TRACE3D tables, CGS units are appropriate. Please refer to the MAD or TRACE3D manuals for more detailed descriptions of the element types used by these programs.

```
create table geometry
(name                    char(20),
value                    char(15)                null,
definition               varchar(90)             null,
comment                  varchar(130)            null)
create unique clustered index geometry_index on geometry(name)
go

create table strength
(name                    char(20),
value                    char(15)                null,
definition               varchar(90)             null,
comment                  varchar(130)            null)
create unique clustered index strength_index on strength(name)
go
```

The geometry and strength tables are used to store the definitions of parameters such as length and magnetic strength of various elements. In cases where these parameters are used frequently in the description of several lattice elements, it is convenient to define a parameter here. There is only one geometry table, but there is the possibility of defining

multiple strength tables corresponding to different operating conditions (injection, ramping, collision, etc.) by using the -T option of DBSF. DBSF uses the columns name and definition. Both of these columns must be filled. The column, value, is optional and can be used to keep track of the values of a parameter if an algebraic expression is inserted into the definition column.

If the user wishes to add extra strength tables, then one must first create these tables using the same SQL as above but with a change in the table name. This name can be as desired since the user must specify the new name in the -T option if she wishes to have that data incorporated in the output. As a general rule we advise the user to first enter all strength definitions in the default strength table. Then after creation of the new table, either fill it by first using the SQL syntax 'insert *new_strength* select * from strength' and then change those strength values appropriate to the new table or by using the bulk copy utilities (or the **strength.in** script described in the User's Guide, section 1.2.2) to load a table with new values from disk. It is VERY important that each 'name' in the various strength tables occur uniformly. Otherwise DBSF may fail.

```
create table magnet_piece
(name                char(20),
type                 varchar(20),
tilt                 varchar(10)          null,
length_defn          varchar(60)          null,
strength_defn        varchar(60)          null,
engineering_type     varchar(20)          null,
comment              varchar(130)         null)
create unique clustered index magnet_piece_index on magnet_piece(name)
go
```

The magnet_piece table is used to define each primitive element in the lattice. MAD 8.1 and TRACE3D element types are accepted. The 'engineering_type' column is especially useful for programs such as MAD or TEAPOT which interpret this extra label (e.g. IR or KVKD) as deserving special attention.

```
create table beam_line
(name                char(20),
elements             varchar(150)         null,
comment              varchar(130)         null)
create unique clustered index beam_line_index on beam_line(name)
go
```

```
create table slot
(name                    char(20),
pieces                   varchar(150)          null,
comment                  varchar(130)          null)
create unique clustered index slot_index on slot(name)
go
create table superslot
(name                    char(20),
pieces                   varchar(150)          null,
comment                  varchar(130)          null)
create unique clustered index superslot_index on superslot(name)
go
```

The beam_line, slot and super_slot tables are used to define the strings of elements in the lattice in a way similar to other accelerator codes (MAD, SYNCH, DIMAD, etc.) which allow this kind of syntax. The list of elements or pieces are entered as strings of names separated by blanks. Commas as separators in the list are not accepted by DBSF.

We have introduced these three beamline tables in the database to provide an extra layer of structure. While accelerator codes do not differentiate between these three, engineers quite often need this distinction. It is common to describe the physical and magnetic properties of a component in the lattice as follows: (drift, magnet, drift). In this case the three elements define a physical slot with the magnet part of the slot having its length defined as the effective magnetic length. So one would introduce this 'line' in the slot table instead of the beam_line table. Such 'slots' can be then be concatenated and entered into the beam_line or super_slot tables.

```
create table name_location
(name                    char(20),
table_name               char(30)              null)
create unique clustered index name_location_index on name_location(name)
go
create table name_alias
(standard_name           char(20),
synch_name               char(5)               null,
comment                  varchar(130)          null)
create unique clustered index standard_name_index on name_alias(standard_name)
go
```

The name_location and name_alias tables are used by DBSF. The name_location table records the element name (beamline or primitive) and the table name it is defined in. As DBSF unravels a beamline into its components, it uses the name_location table as a look up table. The name_location table is filled by a script in the $LAMBDA/sybase

directory (build_nl) which first empties the name_location table and then rebuilds all entries. THE NAME_LOCATION TABLE MUST BE FILLED CORRECTLY FOR DBSF TO RUN PROPERLY. The script **copy.in** (User's Guide, section 1.2.1) will rebuild the name_location table automatically.

The name_alias table is used primarily for SYNCH users. SYNCH has a 5 character limit on names while the database supports up to 20 characters per name. Consequently, if a SYNCH user wishes to truncate names in a specific way, the user should introduce appropriate names for those elements whose names are too long for SYNCH. Otherwise, DBSF will create a unique 5 character name in the SYNCH output if the user does not provide one in name_alias. There is a SQL script in the $LAMBDA/sybase directory (update_name_alias) which should be run when all other tables are filled and which loads the first column (standard_name) of the name_alias table automatically. The first column must be filled in order for DBSF to work with the SYNCH output option.

SECONDARY TABLES

The tables below are to be used whenever a primitive lattice element needs the extra parameters allowed by MAD or TRACE3D. Please refer to the MAD 8.1 or TRACE3D manuals for explanations of the various columns defined below.

NOTE: In order for DBSF to run properly when using the secondary tables, at least the name and type of every primitive element must first be entered in the magnet_piece table. Then DBSF can match the corresponding entry in one of the tables below. It is NOT necessary to use these secondary tables in all cases if the columns in the magnet_piece table are sufficient to describe the element. However, element types such as 'multipole' do require the use of the multipole table.

```
create table bend
(name                char(20),
length               varchar(60)          null,
angle                varchar(60)          null,
tilt                 varchar(10)          null,
quad_strength        varchar(20)          null,
sxtp_strength        varchar(20)          null,
octp_strength        varchar(20)          null,
entrance_angle       varchar(20)          null,
exit_angle           varchar(20)          null,
field_integral       varchar(20)          null,
half_gap             varchar(20)          null,
entrance_curv        varchar(20)          null,
exit_curv            varchar(20)          null)
create unique clustered index bend_index on bend(name)
go
```

```
create table rfcavity
(name                char(20),
length               varchar(60)          null,
voltage              varchar(20)          null,
phase_lag            varchar(20)          null,
harmonic_number      varchar(20)          null,
rf_coupling          varchar(20)          null,
rf_power             varchar(20)          null,
shunt_imped          varchar(20)          null,
fill_time            varchar(20)          null)

create unique clustered index rfcavity_index on rfcavity(name)
go

create table multipole
(name                char(20),
K0L                  varchar(20)          null,
K1L                  varchar(20)          null,
K2L                  varchar(20)          null,
K3L                  varchar(20)          null,
K4L                  varchar(20)          null,
K5L                  varchar(20)          null,
K6L                  varchar(20)          null,
K7L                  varchar(20)          null,
K8L                  varchar(20)          null,
K9L                  varchar(20)          null,
T0                   varchar(20)          null,
T1                   varchar(20)          null,
T2                   varchar(20)          null,
T3                   varchar(20)          null,
T4                   varchar(20)          null,
T5                   varchar(20)          null,
T6                   varchar(20)          null,
T7                   varchar(20)          null,
T8                   varchar(20)          null,
T9                   varchar(20)          null)

create unique clustered index multipole_index on multipole(name)
go

create table closed_orbit_corrector
(name                char(20),
length               varchar(60)          null,
tilt                 varchar(10)          null,
horz_angle           varchar(20)          null,
vert_angle           varchar(20)          null)

create unique clustered index closed_orbit_corrector_index on closed_orbit_corrector(name)
go
```

NOTE: In MAD, closed orbit correctors are typed as HKICKER, VKICKER or KICKER. One of these three types must be entered in the magnet_piece table for closed orbit correctors.

```
create table collimator
(name                    char(20),
length                   varchar(60)          null,
xsize                    varchar(20)          null,
ysize                    varchar(20)          null)
create unique clustered index collimator_index on collimator(name)
go

create table drift
(name char(20),
length varchar(60) null)
create unique clustered index drift_index on drift(name)
go

create table quadrupole
(name                    char(20),
length                   varchar(60)          null,
strength                 varchar(60)          null,
tilt                     varchar(10)          null)
create unique clustered index quadrupole_index on quadrupole(name)
go

create table sextupole
(name                    char(20),
length                   varchar(60)          null,
strength                 varchar(60)          null,
tilt                     varchar(10)          null)
create unique clustered index sextupole_index on sextupole(name)
go

create table octupole
(name                    char(20),
length                   varchar(60)          null,
strength                 varchar(60)          null,
tilt                     varchar(10)          null)
create unique clustered index octupole_index on octupole(name)
go

create table solenoid
(name                    char(20),
length                   varchar(60)          null,
strength                 varchar(60)          null)
create unique clustered index solenoid_index on solenoid(name)
go

create table monitor
(name                    char(20),
```

```
length                  varchar(60)             null)
create unique clustered index monitor_index on monitor(name)
go

create table elseparator
(name                   char(20),
length                  varchar(60)             null,
efield_strength         varchar(60)             null,
tilt                    varchar(10)             null)
create unique clustered index elseparator_index on elseparator(name)
go
```

The following two tables are not part of the MAD or TRACE3D conventions. They are, however, modelled after the collimator element of MAD.

```
create table aperture
(name                   char(20),
shape                   char(1),
xsize                   varchar(30)             null,
ysize                   varchar(30)             null,
centerx                 varchar(30)             null,
centery                 varchar(30)             null)
create unique clustered index aperture_index on aperture(name)
go

create table magnet_size
(name                   char(20),
shape                   char(1),
xsize                   varchar(30)             null,
ysize                   varchar(30)             null,
centerx                 varchar(30)             null,
centery                 varchar(30)             null)
create unique clustered index magnet_size_index on magnet_size(name)
go
```

The aperture and magnet_size tables allow the user to describe the inner and outer transverse dimensions of a lattice element. Since every primitive element in a lattice possesses both inner and outer dimensions, this table in principle has as many entries as there are generic primitive elements. These two tables are currently used by DBSF in the - Cd option which produces the SDS output with the addition of the information in the aperture and magnet_size tables. None of the standard accelerator codes (except perhaps for DIMAD) can use this information at present, so it is not printed in any of the other output options.

The meaning of the columns for these tables is as follows. In analogy with the MAD usage for the COLLIMATOR element, we support an elliptical and rectangular shape for both the inner and outer shapes. The xsize and ysize columns specify the distance to the

edge of the shape from the center of the ellipse or rectangle along the major and minor axis. The columns centerx and centery allow the user to specify the position of the beam with respect to the center of the shape. So, for example, the values -0.025 and 0.01 in the aperture table would put the beam 2.5 cm to the left along the x axis and 1 cm above the center along the y axis of the aperture.

```
create table optics_data
(name                char(20),
t0                   varchar(80)          null,
t1                   varchar(80)          null,
t2                   varchar(80)          null,
t3                   varchar(80)          null,
t4                   varchar(80)          null,
t5                   varchar(80)          null,
t6                   varchar(80)          null,
t7                   varchar(80)          null)
create unique clustered index optics_data_index on optics_data(name)
go

create table survey_data
(name                char(20),
deltax               varchar(40)          null,
deltay               varchar(40)          null,
deltaz               varchar(40)          null,
theta                varchar(40)          null,
phi                  varchar(40)          null,
psi                  varchar(40)          null)
create unique clustered index survey_data_index on survey_data(name)
go
```

Since there are particular cases of accelerator components whose fields are not described simply by the usual magnet types, there is a need for a more generalized description of a tranfer function. A new type of element, the *tcelement* has been defined to satisfy this need. Two new tables in the database have been defined: "survey_data" which contains constants for the "survey" program, and "optics_data" which contains file specifications for SDS format files of the optical transfer function constants.

In the "survey_data" file, each row corresponds to a single tcelement with the seven column entries `name, deltax, deltay, deltaz, theta, phi, psi`. The first column contains the name of the element, and the other six constants describe the effect on the design trajectory as specified in the documentation for the MAD program.

The "optics_data" file also contains a single row for each tcelement. Each row contains the `name` entry followed by eight fields (`t0, t1, t2, t3, t4, t5, t6, t7`) for specifying the file names for the SDS files. At present only the `t1` entry is used for the linear transfer matrix elements in the program "twiss". (For the RHIC database these files reside in

```
/usr/local/Holy_Lattice/tcelements
```

The tcelement SDS files contain data in the form of a C structure:

```
struct
char indices[8];
double value;
element[];
```

The eight indices run from 0 to 5 corresponding respectively to coordinates (x, x', y, y', z, δ). Only nonzero coefficients are stored in the SDS file.

The optical transfer function may be desribed by the Taylor series

$$X_i(s_1) = T_0^i + \sum_{j=1}^{6} T_1^j X_j(s_0) + \frac{1}{2}\sum_{j=1}^{6}\sum_{k=1}^{6} T_2^{jk} X_j(s_0) X_k(s_0) + \frac{1}{3!}\sum_{j=1}^{6}\sum_{k=1}^{6}\sum_{l=1}^{6} T_3^{jkl} X_j(s_0) X_k(s_0) X_l(s_0) + \ldots,$$

where $\mathtt{T}_0$ is a constant offset vector given by the "t0" file, $\mathbf{T}_1$ is the Jacobian matrix given by the "t1 file, and the higher order tensor $\mathbf{T}_n$ is given by the "$\mathtt{t}_n$" file.

Trace3d TABLES

The following tables are used specificly by TRACE3D. Please refer to the TRACE3D manual to understand the columns in each table.

```
create table thinlens
(name                char(20),
xfocal               varchar(20)          null,
yfocal               varchar(20)          null,
zfocal               varchar(20)          null)
create unique clustered index thinlens_index on thinlens(name)
go

create table pmq
(name                char(20),
bfld_grad            varchar(20)          null,
length               varchar(20)          null,
inner_radius         varchar(20)          null,
outer_radius         varchar(20)          null)
create unique clustered index pmq_index on pmq(name)
go

create table doublet
(name                char(20),
bfld_grad            varchar(20)          null,
length               varchar(20)          null,
dist_between         varchar(20)          null)
create unique clustered index doublet_index on doublet(name)
go
```

```
create table triplet
(name                char(20),
bfld_grad_outer      varchar(20)     null,
length_outer         varchar(20)     null,
dist_between         varchar(20)     null,
bfld_grad_inner      varchar(20)     null,
length_inner         varchar(20)     null)
create unique clustered index triplet_index on triplet(name)
go

create table edge
(name                char(20),
angle                varchar(20)     null,
radius               varchar(20)     null,
gap                  varchar(20)     null,
fringe_factor1       varchar(20)     null,
fringe_factor2       varchar(20)     null)
create unique clustered index edge_index on edge(name)
go

create table rfgap
(name                char(20),
eff_gap_voltage      varchar(20)     null,
phase                varchar(20)     null,
eg_flag              varchar(20)     null,
dW_flag              char(1)         null,
harmonic             char(1)         null)
create unique clustered index rfgap_index on rfgap(name)
go

create table rfqcell
(name                char(20),
V_div_rsquared       varchar(20)     null,
AV                   varchar(20)     null,
length               varchar(20)     null,
phase                varchar(20)     null,
type_cell            char(1)         null)
create unique clustered index rfqcell_index on rfqcell(name)
go

create table tank
(name                char(20),
accel_grad           varchar(20)     null,
length               varchar(20)     null,
phase                varchar(20)     null,
iden_cav             varchar(20)     null)
create unique clustered index tank_index on tank(name)
go
```

This concludes the description of the SQL code needed to create the lattice database tables. Once the user has filled the tables appropriately, access must be granted so that

other users and applications can use the data. The following SQL provides the necessary access to the above tables. In particular the user, 'lattice_reader', is added to each lattice database in order to allow DBSF to function properly.

```
sp_adduser guest
sp_adduser lattice_reader
go
```

sp_adduser is a SYBASE stored procedure which adds the given user to your database. The user 'guest' is a special user name in SYBASE. The addition of this user allows all SYBASE users in the 'public' group to access your tables if you grant read permission to the public group. The guest user eliminates the need to add each SYBASE user to your database.

```
grant select on geometry to public
grant select on strength to public
grant select on magnet_piece to public
grant select on beam_line to public
grant select on name_location to public
grant select on slot to public
grant select on name_alias to public
grant select on rfcavity to public
grant select on bend to public
grant select on closed_orbit_corr ector to public
grant select on multipole to public
grant select on collimator to public
grant select on drift to public
grant select on quadrupole to public
grant select on sextupole to public
grant select on octupole to public
grant select on solenoid to public
grant select on monitor to public
grant select on elseparator to public
grant select on aperture to public
grant select on magnet_size to public
grant select on optics_data to public
grant select on survey_data to public
go
```

TRACE3D permissions

```
grant select on thin_lens to public
grant select on pmq to public
grant select on doublet to public
grant select on triplet to public
grant select on edge to public
grant select on rfgap to public
grant select on rfqcell to public
grant select on tank to public
go
```

The above statements grant select (i.e., read only) privilege to each of the lattice database tables. All the sql described above is contained in the main_setup script contained in the $LAMBDA/sybase directory. The user need not execute any of the above sql statments separately if this script is used to create the database tables and permissions.

3.2 Creating a lattice database

1. To create an empty database, "DATABASE_NAME"
   Enter the Data Workbench by typing **dwb** and logging in as "sa", in the "master" database.
   Run the SQL command **create database DATABASE_NAME** (and be prepared to wait a minute or less)
   If desired, (YES !) enable bulk copying in and out of database tables by performing the commands in section 2, below.
   Change the database context to DATABASE_NAME
   Run the SQL command **sp_changedbowner OWNER_NAME** to change the owner of the database to the login name of the appropriate owner.
   Create the tables, add appropriate users, and grant appropriate permissions, by performing the commands in section 3, below.
2. To enable bulk copying in and out of tables
   Enter the Data Workbench by typing **dwb** and logging in as "sa", in the "master" database.
   Run the SQL command **sp_dboption DATABASE_NAME, bulkcopy, true**
   Run the SQL command **sp_dboption DATABASE_NAME, trunc, true**
   Change the database context ("Set DB Context") to DATABASE_NAME
   Run the SQL command **checkpoint**
3. To create lattice tables, add users, and grant permissions
   Temporarily copy the isql script $LAMBDA/sybase/main_setup. to your local directory.
   Edit the first line of your copy of the script, to name the database of interest, which you must own. For example use **DATABASE_NAME**
   Invoke the script by saying **isql -U username -P password < main_setup.**

3.3 isql scripts

Although it is possible to upload ASCII files of pure **SQL** commands into the Data Workbench, it is usually more convenient to modify the file to become an **isql** script that

can be run from the regular computing environment. For example, to run an isql script $LAMBDA/sybase/script_name.i, first copy the script into a local directory and then edit the first line to name a database that you own. For example, edit **use pta** to become **use my_database_name.** Then say

isql -U username -P password < script_name.i

The following useful **isql** scripts are available in $LAMDA/sybase.

script name	**brief description**
trace3d_setup.	Create optional tables for linac description, using Trace3D formats.
build_nl.	Rebuild the "name_location" table that serves as an index table to **dbsf**. This is especially useful if a small number of changes have been performed inside the Data Workbench - that is, without having to use **copy.in** or **copy.out**.
revoke_all.i	Revoke all permissions on all tables to public
main_setup.	Create all tables, add lattice_reader as a user, grant select permissions to public. See section 3, above.
busql	creates bus table information for wireup program.
create_view_rules	creates tables needed to use view option in dbsf.

4. REFERENCES

A Standard Input Format agreement.

5. APPENDIX

PLOT_TWISS(1) USER COMMANDS PLOT_TWISS(1)

NAME

plot_twiss - plots lattice functions from program "twiss"

SYNOPSIS

plot_twiss [-s]

OPTIONS

plot_twiss recognizes the standard Xtoolkit switches, as well as the -s switch which indicates that the input is to come from shared memory.

DESCRIPTION

This program reads the output SDS file "Twiss" from the program "twiss". With the -s switch it reads from shared memory; whereas, without the -s switch it reads "Twiss" from the current path.

The following keys may be entered either into the window or into "stdin". (Note that "stdin" requires a carriage return after the character.)

"a": unzoom to the whole lattice
"h": print a help message to "stdout"
"p": output plot to PostScript file "plot_twiss.ps"
"q": quit
"r": reread the data and plot again

"plot_twiss" draws the horizontal and vertical beta functions in the upper portion of the window and the horizontal and vertical eta functions in the lower portion. A schematic of the magnetic elements are shown in the middle of the window. Clicking on one of the elements will select that element (highlit by red) by writing the element number to "stdout"; this can be used to generate a pair of glish events for example to show the input and output parameters with "sid". By having glish send an "r" to "stdin", "plot_twiss" may be triggered to replot the data after a change has been made.

As a default, the left mouse button will zoom the left boundary of the plots if the cursor is in the upper or lower portions of the window (not in the middle "lattice" section). The middle button will zoom the right side, and the right button unzooms the plot (same as "a").

X-RESOURSES

The user can customize "plot_twiss" with the following resourses in the ".Xdefaults" file:

```
plot_twiss.background: black
plot_twiss.geometry: 300x200
*plot_twiss.translations: "#augment\0
     <Btn1Down>,<Btn1Up>: zoomleft()\n\
     <Btn2Down>,<Btn2Up>: zoomright()\n\
```



```
     <Btn3Down>,<Btn3Up>: unzoom()"
```

Setting the background color stops the slightly annoying white flash when the window is first mapped; the background is set to black eventually, so you shouldn't waste time try-

ing to change the color. The other settings can be changed to whatever you want. Southpaws may want to swap the first and third buttons, if they have remapped the buttons via something like:

```
xmodemap -e "pointer=3 2 1"
```

SHARED MEMORY

In order to place files into shared memory you need to do the following:

1) Create a directory "shardat" in your home directory, "~/shardat".

2) Create empty files in "~/shardat" for each file to be placed into shared memory. For example, "Twiss" for the "twiss" output file and "ags_to_rhic.a2yellow" for the flat-SDS file "ags_to_rhic.a2yellow". (You can use "touch" to create empty files.)

3) Generate the SDS files in your work area, e.g.,

```
setenv DBSF_DB ags_to_rhic     #to tell dbsf which database
dbsf -C a2yellow               #create "ags_to_rhic.a2yellow"
f2sm ags_to_rhic.a2yellow      #copy it to shared memory
twiss -s ags_to_rhic.a2yellow #run twiss in shared memory
```

You could also run "twiss" without the "-s" switch and then use "f2sm" to copy "Twiss" to shared memory.

The following programs are useful for dealing with shared memory files:

```
wshm                -- lists files in shared memory
srm  filename       -- delete a file from shared memory
clearshm            -- delete all files from shared memory
f2sm filename       -- copy filename into shared memory
sm2f filename       -- copy filename from shared memory
sid -s filename     -- Salty's SDS spreadsheat
sid -w -s filename -- Run "sid" in edit mode
smd [-s] filename [object #] -- ascii dump of SDS file.
```

SEE ALSO

The LAMBDA twiss manual

BUGS

If I knew of any bugs, don't you think I'd fix them? (Ha! Ha! Ha!)

There are however a few undocumented features which I won't describe, so that they may remain undocumented. See if you can find them.

AUTHOR

Waldo MacKay

COPYRIGHT

Copyright (C), 1993 by the author.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose so long as it remains FREE, provided that the above copyright notice appear in all copies and that both that copyright notice and

this permission notice appear in supporting documentation.