The Integrated Scientific Tool Kit:
An Overview of the System and its Components

Ericka Lutz and Chris Saltmarsh

August, 1992

Table of Contents

Introduction to the Integrated Scientific Tool Kit

The Integrated Scientific Tool Kit (ISTK) is a computer software tool and component kit developed for use by experimentalists. ISTK's development has originated in software design for particle accelerators but its uses are applicable to many experimental physics control and analysis systems.

ISTK's three main components are designed to manage major areas of software systems which are often not sufficiently served by commercial or industrial products. By first looking at the tasks involved to carry out a project, it is easy to determine which ISTK components are appropriate for the tasks at hand.

SDS (the Self-describing Data Standard) facilitates data flow: the movement of data between computers and between applications. If any data will be used in more than one process, it should be put in an SDS.

The Communication Systems (Glish and Communistks) manage the control flow of operational sequences. The kind of control flow performance needed, between elements of a single application or between applications, determines whether Communistks or Glish (the internal and external control flow components) should be used.

GLISTK (the Graphic Library for the Integrated Tool Kit) handles the interface between people and computers.

Some of the currently available ISTK tools include:

sid, an interface to numeric and textual data that provides customized access and editing mechanisms.
latview, a graphical view of accelerator lattices that is able to signal user queries to other programs.
conv, a program that converts binary data files to different machine architectures.
gasp, a pre-processor for basic SDS program writing.
The Glish executive, a program to carry-out complex and adaptive multi-processor sequencing.

The developers of ISTK are involved in the ongoing process of developing multi-use software tools that can be assembled and modified for the user's own purposes. ISTK is also a collaborative project. Tools and ideas are being developed at the Superconducting Super Collider Laboratory, Lawrence Berkeley Laboratory, FermiLab and other physics laboratories world-wide. People are encouraged to add their own multi-use ISTK tools to the existing ISTK libraries and utilities.

Document Format

This document exists on two "tracks". Those readers desiring a general overview of the Integrated Scientific Tool Kit, its approach and its components, should read Track One. Those desiring more in-depth, technical information should supplement Track One with Track Two.

A Cost-Effective Approach

There is a problem inherent in designing software: design and costing procedures often become a large part of a project. One common dilemma is that, unless a software project in development is already well understood, there is considerable financial risk in the early stages yet full understanding of software needs often comes only once the software development has been done. This dilemma makes it difficult and expensive to make a commitment to full software implementation.

The ISTK components have been designed to aid in the preliminary stages of product or system delivery (beyond the basic proof-of-principle tests). Using the ISTK components, ideas can be explored, explained, made more precise, and discarded (if necessary) without great additional cost.

The ISTK system can quickly respond to changes because the realm of application in which it was born (technically challenging experimental control and analysis) is always in such a phase of development. In large physics experiments, software flexibility is required at every stage; by the time a system has been "hardened" into a final form, other factors (equipment, realm of applicability) have changed and the software must be altered to meet the new needs.

ISTK's strengths are its flexibility and its faculty in filling the gaps between available software programs. ISTK cannot replace all software but it can be used as an excellent supplement to existing commercial software. In a constantly fluctuating experimental situation, ISTK's flexibility provides a great advantage. In a market where factors are more stable, the ISTK use-approach and design emphasis on flexibility for the user can also give significant advantages during the appraisal and start-up phases.

ISTK provides a variety of tools for building various kinds of systems. The components can be used as prototyping tools to create applications in a short period of time, for experiment preparation (simulations, test experiments, etc.), controls systems, and for preliminary data analysis. Much of the ISTK system is sufficiently hardened to form the basis of production code decisions; decisions that can be made effectively only after a job analysis process.

Deciding Which Components to Use

The appropriate way to use ISTK is to break each overall job into basic, general tasks. In order to decide which tools are needed for a particular job, the tasks should be broken into generic bits which are as dissociated from the particular application as possible. Some of the possible tasks inherent in a jobs might be: organizing data, displaying graphics, listing datasets, etc. This conceptual analysis includes determining the needed processes, defining the flow of data between these processes, and defining how these processes should be connected by control flow. All of this analysis should occur *before* any implementation takes place. The design of the major ISTK components; SDS, the data discipline and Glish, the control-flow manager, are, in a sense, a bottom-up response to top-down and object-oriented analysis techniques.

Implementation Decisions

The ISTK components can be used for developing large scale systems as well as for designing simple and complex single applications. The conceptual approach is basically the same; only the implementation differs. ISTK tools are not specific to any one application. Strong attempts have been made to insure that the tools are well-designed enough to be useful, yet abstract enough to be used in a variety of circumstances.

CAVEAT:

It is important to remember that the ISTK tools and techniques do not, in themselves, solve problems. Their role is to ease the ability of skilled people (programmers, engineers, physicists) to encapsulate and pass on their problem-solving experience to end-users, end-users who do not want or need the depth or breadth of knowledge contained in the software structures in order to perform their tasks. In this, ISTK is like all other software systems. To pretend otherwise is wrong; to expect more will lead to disappointment.

Approach Track 2

More About ISTK and the Unix-Style Approach

The ISTK package contains general all-purpose applications; components that can be used alone, together, or as a supplement to pre-existing software to aid in clarifying and organizing experimental data and operations. The ISTK system of using inter-connected components owes a large debt to the principles used to build Unix, that is, independent modules each designed to do a single job well, working together in conjunction with each other. The result of this approach for Unix was an immensely capable and coherent operating system.

ISTK's various programs are not connected until they are meshed together in the net in any way that the operator chooses. The ISTK applications can be thought of as Tinker Toys, components which remain the same, hooked together by changeable, flexible, connections. By merely altering the way the components are connected, the user can build elaborate programmatic structures without making internal program changes.

ISTK extends the concepts of the Unix system. Whereas Unix is a programming environment, ISTK extends it to components that provide great flexibility for people doing physics experiments. Conceptually, the SDS data discipline and the Glish control flow description parallel the general rules of the Unix operating system and allow the building of general purpose tools.

ISTK Communication Systems

Glish and Communistks

The ISTK Communication Systems, Glish and Communistks, are the ISTK components in charge of control flow within and between applications. Within ISTK applications, connections are managed by the Communistk, the ISTK communication class. Between applications, ISTK connections are managed by Glish. Conceptually, the Glish and Communistk connections are similar. However, their implementation varies greatly.

What are Communistks?

Communistks are the communication class for the ISTK libraries. A Communistk is an object which focuses on a value and has a list of other data objects to notify when this value, or the Focus itself, changes. Communistks exist only in the object-oriented languages. They enable and simplify communication between objects in a program and, by using the Glish connection, between different programs within a processor and between programs running on physically distinct processors.

What is Glish?

Glish is both a rule-based language to describe high-order sequencing and an executive to carry out this sequencing. The Glish language explicitly states the control flow so that it can be managed and properly understood. The Glish Executive is also a multi-task controller and inter-task communication system.

The Glish system separates the high-level control from the mechanical, pre-determined software and hardware operations and connects the control flow to an up-front sequencing language that coordinates interdependent sequences of widely distributed processes. Because of this separation, the Glish sequence of operations is protected from non-vital hardware failure.

Glish enables tasks to be broken into small components for clarity and organization. The components run as small programs orchestrated by Glish into a larger program with a common set of rules and conditions. Any program, even one completely unaware of Glish, can be run and monitored by Glish.

Glish combines the benefits of computer sequencing with the flexibility of on-line human modification. It allows quick, immediate input while freeing people from redundant or petty operating decisions.

Communications Track 2

The Glish Language and the Glish Executive

Glish establishes, in a few words, rules on how to respond to an infinite set of circumstances. To do this, Glish has two parts, the Glish language and the Glish executive. The Glish Language is used to express the sequencing and the Glish Executive carries out that sequencing. The user establishes the cause and effect relationships by writing the Glish script. The Glish Executive then orchestrates and sequences events between small programs and organizes them into a larger program with a common set of rules and conditions. The Glish Executive is much like a switchboard operator who establishes the connections between appropriate "parties", allowing the transfer of information.

(The relationship between the Glish language and the Glish Executive is paralleled by the relationship between SQL as a data definition standard and the DBMS's that implement all or part of that standard, and the relationship between FORTRAN as a language standard and compilers that implement all of part of that standard.)

Glish is not restricted to one Executive, and in different circumstances, different Executives may be appropriate. For example, in a particle accelerator, there is a range of control flow, from hard-wired power supply settings, through theoretical and mathematical abstractions, to subjective decisions (like determining the full validity of the results). At times, a combination of Executives might be installed on the same system; the high-level Executive interfacing with the human operators and the low-level real-time system Executive in charge of the hard-event driven functions, both using the same sequencing rules, expressed in the same language.

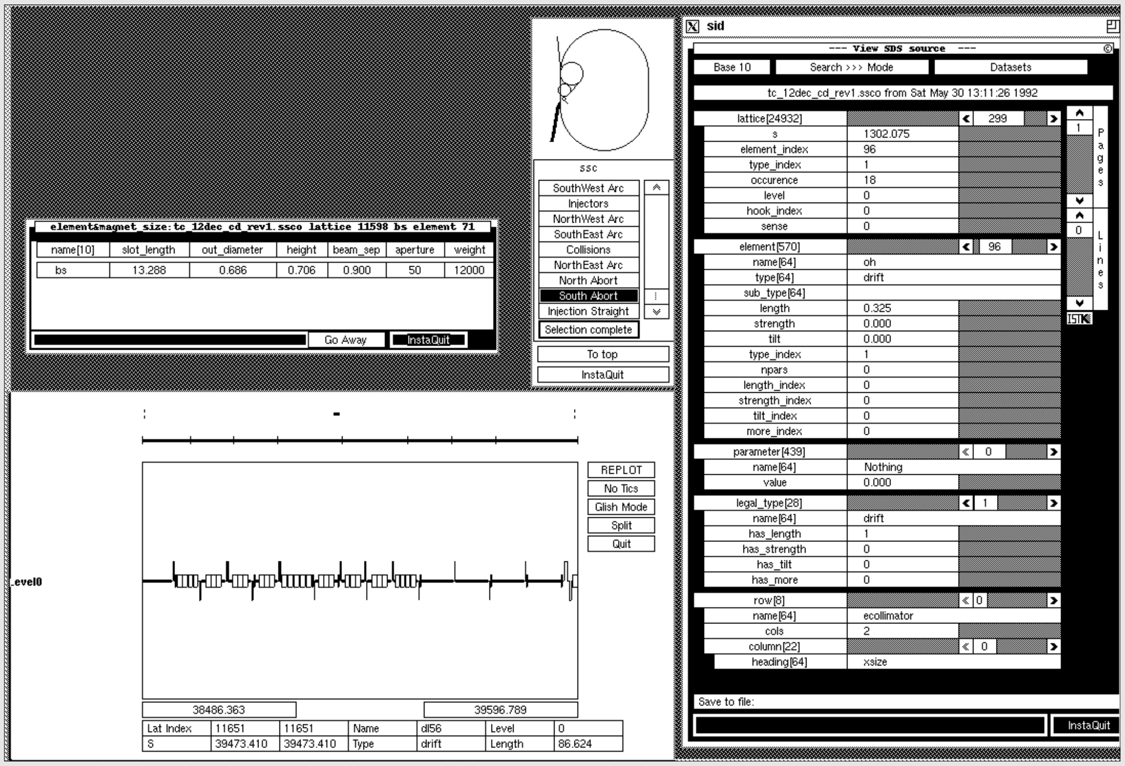
Communications Track 2

```
oswald := client("oswald -T  -geometry +1+1");
sscsel := client("sscsel  -geometry +660+1");
catch := client("Ccatch  -geometry +190+380");
sid := client("sid");
whenever sscsel->SdsFile do
{
  oswald->SDSFILE($value);
  sid->Display($value);
}
whenever oswald->MoveTo do
{
  catch->ElementIndicator($value);
}
whenever sscsel->SRange do
{
  oswald->SRange($value);
}
```

Glish Script

The Glish language uses some unusual constructs ("whenever", "if and whenever" etc.) to handle sequencing situations. A simple description is written to describe what is expected to occur, the events that should be generated by those occurrences and under what conditions those events should be channeled to other processes. Because Glish is rule-based, this description of conditions can be written in any order. When the conditions defined in the Glish script are met, Glish sends named events to specified processes ("clients"). Glish can start and stop its clients and can provide inter-process communication between various clients. Any two (or more) processes running anywhere on the network can use Glish to communicate.

Communications Track 2



element&magnet_size:tc_12dec_cd_rev1.ssco lattice 11598 bs element 71

name[10]	slot_length	out_diameter	height	beam_sep	aperture	weight
bs	13.288	0.686	0.706	0.900	50	12000

38486.363 39596.789

Lat Index	11651	11651	Name	dl56	Level	0
S	39473.410	39473.410	Type	drift	Length	86.624

--- View SDS source ---

tc_12dec_cd_rev1.ssco from Sat May 30 13:11:26 1992

lattice[24932]		299
s	1302.075	
element_index	96	
type_index	1	
occurence	18	
level	0	
hook_index	0	
sense	0	
element[570]		96
name[64]	oh	
type[64]	drift	
sub_type[64]		
length	0.325	
strength	0.000	
tilt	0.000	
type_index	1	
npars	0	
length_index	0	
strength_index	0	
tilt_index	0	
more_index	0	
parameter[439]		0
name[64]	Nothing	
value	0.000	
legal_type[28]		1
name[64]	drift	
has_length	1	
has_strength	0	
has_tilt	0	
has_more	0	
row[8]		0
name[64]	ecollimator	
cols	2	
column[22]		0
heading[64]	xsize	

Save to file:

A screen dump of the interfaces controlled by the Glish script of (). Accelerator regions are chosen by the graphical selector (center top) and displayed graphically (bottom left) and textually (right). Selection of a magnetic devicee from these latter two displays feeds an SQL query to the main database. The results of this query, giving more detail on the device, are shown at top left. Each display is from a seperate program. They are controlled by Glish events.

Internal communication also exists within each display. In the display at right, for instance, the lattice structure 299 (at the top of display) is connected to the appropriate element type 96 (just below) by Communistks. These internal connections may also be sent out of each program via Glish.

Communications Track 2

Tuning an Accelerator Beam

The example of tuning a particle accelerator beam demonstrates how the ISTK sequencing components, (Glish and Communistks) allow human intervention in the computer sequences that run the machine without interfering with the overall sequence. The example below shows how processes can be internally "re-wired" using Communistks and externally "re-wired" between each other using Glish commands.

The Tune

Setting the "tune" of a particle accelerator is crucial to its performance. When a beam of particles spins around in an circular accelerator, magnetic fields try to focus it into the middle, forcing that beam to oscillate. This beam of particles oscillates at a particular frequency (number of times around the ring), based upon the strength of the focussing magnetic field.

The exact number of oscillations the beam performs per cycle is vital, as various things can conspire to make the oscillations bigger and bigger, eventually destroying the beam. An accelerator is never, for instance, set to an integer tune. A beam set to an integer tune will oscillate exactly that integer number of times and cycle back in exactly the same place in exactly the same direction. If there are any imperfections in the magnetic field (a situation impossible to avoid), the beam set to an integer tune will be kicked slightly sideways by these imperfections every time it cycles the accelerator, and very soon the beam will be kicked completely out of place. (Imagine the actions of a child on a swing pushed by somebody else. If the pusher pushes at the same place in every oscillation, even if the pushes don't have much force, the oscillations will get bigger and bigger and bigger, until the child falls off. If, however, the child sitting on the swing is hit randomly, she and the swing will stay at the bottom of the curve, oscillating only slightly.) It is therefore vital to set the beam to a non-integer tune so that the first time the beam cycles the ring it will return to the center of the ring, the second time around it will have gone many oscillations plus a little bit, etc. This argument is true for oscillations that repeat after 2, 3, 4, ... times and can get quite complex in interactions with other forces. The principle is the same in all cases. The tune must therefore be able to be measured and adjusted slightly as the accelerator runs.

Measuring the Tune

A beam pick-up is located at one point in the accelerator ring and measures the beam position on successive turns. (In this example, the important part of the measurement is the post-decimal number. The integer part of the number of oscillations is not important for this example.) From these measurements a graph is obtained. The graph will show a wave pattern that may damp down as the oscillations get smaller and smaller or it may increase in magnitude, indicating unstable operation.

A Fourier transform is performed (usually using a mathematical technique called a Fast Fourier transform or FFt) to translate the wave form into a graph of a musical tone, showing pitch, movement and timbre. If a beam movement is analogous to the guitar string motion when the guitar string is plucked, the FFt transform (no information is added or deleted) is the mathematical representation of what the human ear apprehends from the guitar string motion; not the change in position of the string, but a perception of the range of speeds of movement.

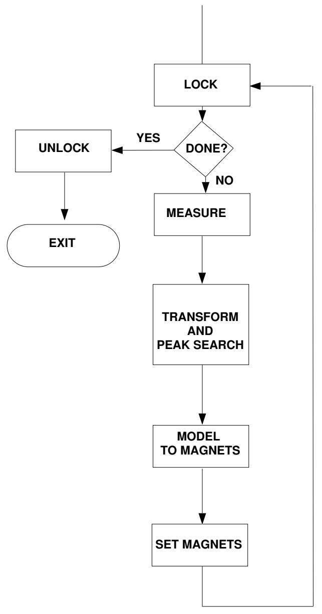
The beam should not measure as a pure tone, as a too-exact tone is at risk for becoming unstable. The standard beam will plot with some noise at low frequencies, a big peak (not a delta function, but a band of peaks), and more noise at the high test frequencies.

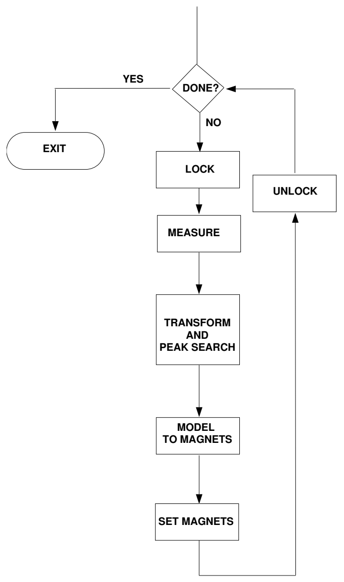
Doing a peak search to obtain the top number of the peak (fitting a curve to the highest points), the number of the tune is finally obtained. After the tune number is obtained, changes can be made to it, the power supplies can be altered accordingly, and the tune remeasured.

Correcting the Tune

As the particles are accelerated toward top energy (the sped at which the experiments will work), the magnetic bending and focussing must change in concert to control the progressively "stiffening" beam. The tune of the accelerator may change as the beam is ramped up from low injection speed to high field where the experiments can take place. As the beam accelerates, the tune is measured every sixty milliseconds and these measurements are plotted time against tune. The graph of the tune should show smooth, well-understood and expected movements. At times, small glitches around the curves occur and this is one place where the power supplies often need to alter the tune to flatten the glitches. Once the particle beam has been accelerated up to speed, other small adjustments may need to occur as well.

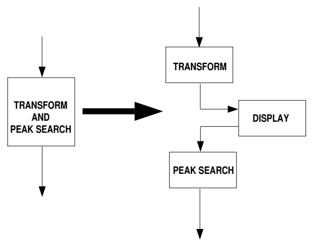
Because these tune measurements need to be taken so often to get an accurate picture of the beam, is appropriate that computer programs (rather than people) be designed and implemented to do these measurements automatically with human intervention only when necessary. Adjustments to the beam also occur at rapid speeds and it is appropriate that computer programs perform these other functions as well.





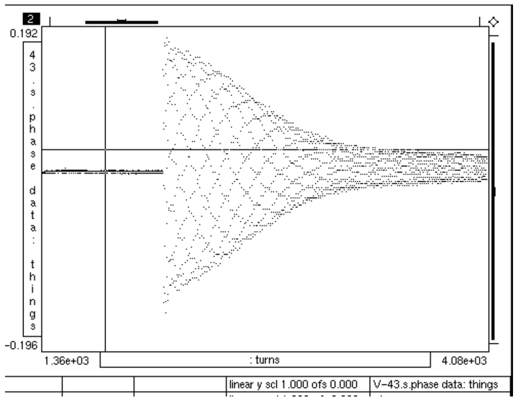
This is a highly simplified block diagram illustrating tune measurement and feedback. One problem: as the machine is locked while an extensive group of measurements is done, operators may be unable to work for a considerable time...

... a change at the Glish script level will allow other users to touch the machine between measurements...

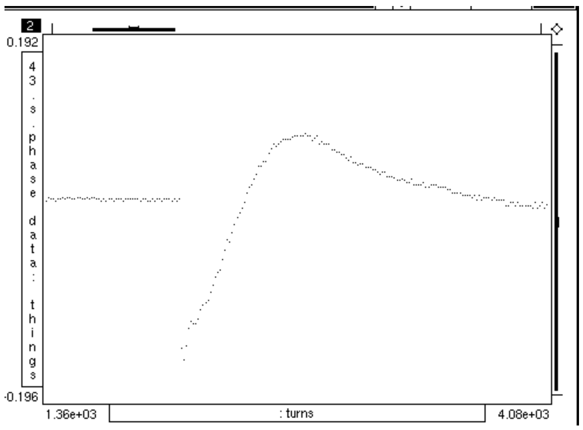


...to break control within a program (here to inspect data halfway through analysis), one must inform internal communication links to reattach to outside processes. This is also possible dynamically and without recompilation.

Tuning the Accelerator Beam



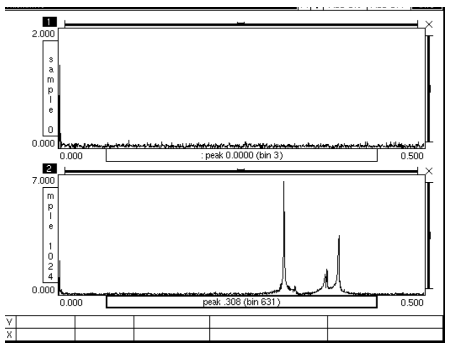
Tevetron beam oscillating back to central position after a sideways kick...



...the graphical browser "kasper" has been asked to plot only every thirteenth point. The plot demonstrates that the beam is on a thirteenth order resonance...

transform of data before kick →

transform of data after kick →



...a Fourier transform shows a strong peak at a fractional tune of 4/13.

Altering Sequences

Both ISTK sequencing components, Glish and Communistks, have the flexibility of allowing sequences to begin operation with a small number of rule-based functions and having other control flow statements injected later. They also allow the functions to be disassociated from each other so that if one part of the sequence goes wrong, the entire sequence will not crash. They allow human intervention in the computer sequences that run the machine without interfering with the overall sequence.

Using Communistks to Alter the Beam Tune

In the instance of the particle accelerator's tune, a program can be written to take beam measurements and transform them into a graph. Sometimes though, for instance, the original beam measurement is inaccurate and the tune/time analysis will show a wayward analysis. In this case, the FFt might show "fuzz" and this noise will lead to a false peak during the peak searching analysis. If this occurs, the accelerator operators need access to the program during the time between the FFt and the peak search, a time where there is a single control flow. Communistks enable a person to plug in a viewer and control panel at this point. Only when the operator approves of the FFt will the peak search continue.

Using Glish to Alter the Beam Tuning Sequence

It is necessary to prevent other input during the period of time between measurement of the beam tune and alteration of the power supplies that control the beam tune. For this reason, the machine must be locked during these procedures. The long length of time taken by locking the machine (thus monopolizing it), performing the measurements, altering the power supply, remeasuring, re-altering the power supply, and completely correcting the beam before unlocking the machine can become a problem when other people and functions need access to the machine. Glish enables the "wiring" to be changed with a statement of rule change, moving the lock/unlock functions into the loop cycle so that there is time between each alteration cycle for others to "play through".

Communications Track 2
Communistk Elements: The Focus and The Constrainer

The Focus

The Focus of a Communistk is essentially a "Thing of Interest" to a program object; in computer terms, a memory location into which a value is written. The Communistk is responsible for policing what happens to that Focus and for routing advice of any changes to the Focus' value to data objects which have expressed an interest in the Focus. The Constrainer class holds constraint information about the value of the focus.

When there has been a change to the Focus, the Communistk informs data objects which have an interest in the Focus about the changes. These data objects don't know or care which or how many data objects may change the Focus. They are only interested in knowing if it has been changed. The data object which changes the value of the Focus does not notify any other data objects of its action; it is the data objects who wish notification which ask the Communistk to inform them of any changes to the Focus.

Communistk Elements

The Present Datum device is thus notified by the Communistk of a change in the value of the Focus. It does not change the data, it only presents the data.

When the change datum device, the "catcher", is given a change request from a person, it requests information from a separate data source that holds information about the name and type of the focus.

The change request is then filtered through the Communistk and the Constrainer. Between the Communistk and the Constrainer is a forced internal link.

Communistk Constraints

Any changes to the focus are filtered through the Constrainer which holds pre-established allowable values; minimums, maximums, etc. The information about these constraints is held in a separate data source. Responsibility for changes to the focus lies with the Constrainer, the low-level wrapper. The change-datum devices do not need to take on that responsibility.

The Communistk creator, when creating space for the Communistk's Focus or when attaching to the Focus, can assure that attempts to write in a value of an incorrect value type will be blocked. Additional constraints on that value (hard minimum and maximum values, warning and alarm values, restrictions on single-hit changes, etc.), may be placed using the Communistk Constrainer class. These constraints are associated with the Focus itself, not with the data object that is inputting the value change. Any attempt to change the value of the Focus are properly checked and illegal actions will be thrown out by the Communistk. Such actions may be monitored without intervention of the object which tried the modification.

Change of Focus

In addition to the change in a Focus' value, the Focus itself may move from one data object to another. An analogy for this might be the "daily specials" list at the local restaurant:

Joanne's Restaurant has a list of dishes served only on Mondays (roast beef with potatoes, vegetarian casserole, shrimp salad), a list of dishes served only on Tuesdays (Chef's Salad, Sloppy Joe with boiled carrots, calves liver saute), etc. These lists remain the same; every Monday the specials are the same as every other Monday. Joanne is lazy, she doesn't like to rewrite the Specials Board every day, so she has seven boards in the closet, each one listing the specials for one day of the week.

Joanne's Special's Boards are analogous to the Foci. Foci value (like the daily lists) remain the same, but a move in the Focus resets which value is being looked at (Monday's, Tuesday's or Wednesday's Board). There are a number of methods which do no changing of Focus value but simply declare to all interested parties that the data Focus has shifted.

The Self-describing Data Standard

SDS, the Self-describing Data Standard, is the ISTK component that facilitates data flow: the movement of data between computers and between applications.

In fundamental terms, SDS is a description of the structure of a selected quantity of data. SDS binds a descriptive information header to this data; this layer of information helps facilitate data manipulation and data transfer between processors of diverse architecture and between various applications within a computer.

By using the SDS description, every application in the system, whatever its "world view", has the ability to utilize the same data. (SDS does not control or determine data formats; the person generating the data determines the appropriate format.) For applications designed to analyze general forms of data in general ways (displays, Fourier transforms, etc.), SDS aids by providing a full description of the data to be analyzed, and for applications designed to analyze specific types of data in specific ways, SDS aids by filtering out inappropriate data.

SDS' strength (like that of all the ISTK components) is in its ability to fill the gaps between other available software. SDS has been created to provide a common link between data management techniques in an experimental situation where it is effectively impossible to replace the myriad of processors and languages with a single system.

Transferring data between applications and between processors can present problems, especially for people in the experimental world who need a great deal of flexibility. Many of these problems are based on differences in data structuring and memory allocation in different systems.

SDS facilitates the data transfer between processors of different architecture. Perhaps a good way to think of SDS is as "glue", a substance that binds these components (systems, languages, applications) together at the level of bits and bytes, the level of memory allocation and data structuring.

SDS operates on a higher level of data description. One of the important SDS data description features is the SDSListk, a type of persistent list that requires no duplication of data items and no manual memory allocation.

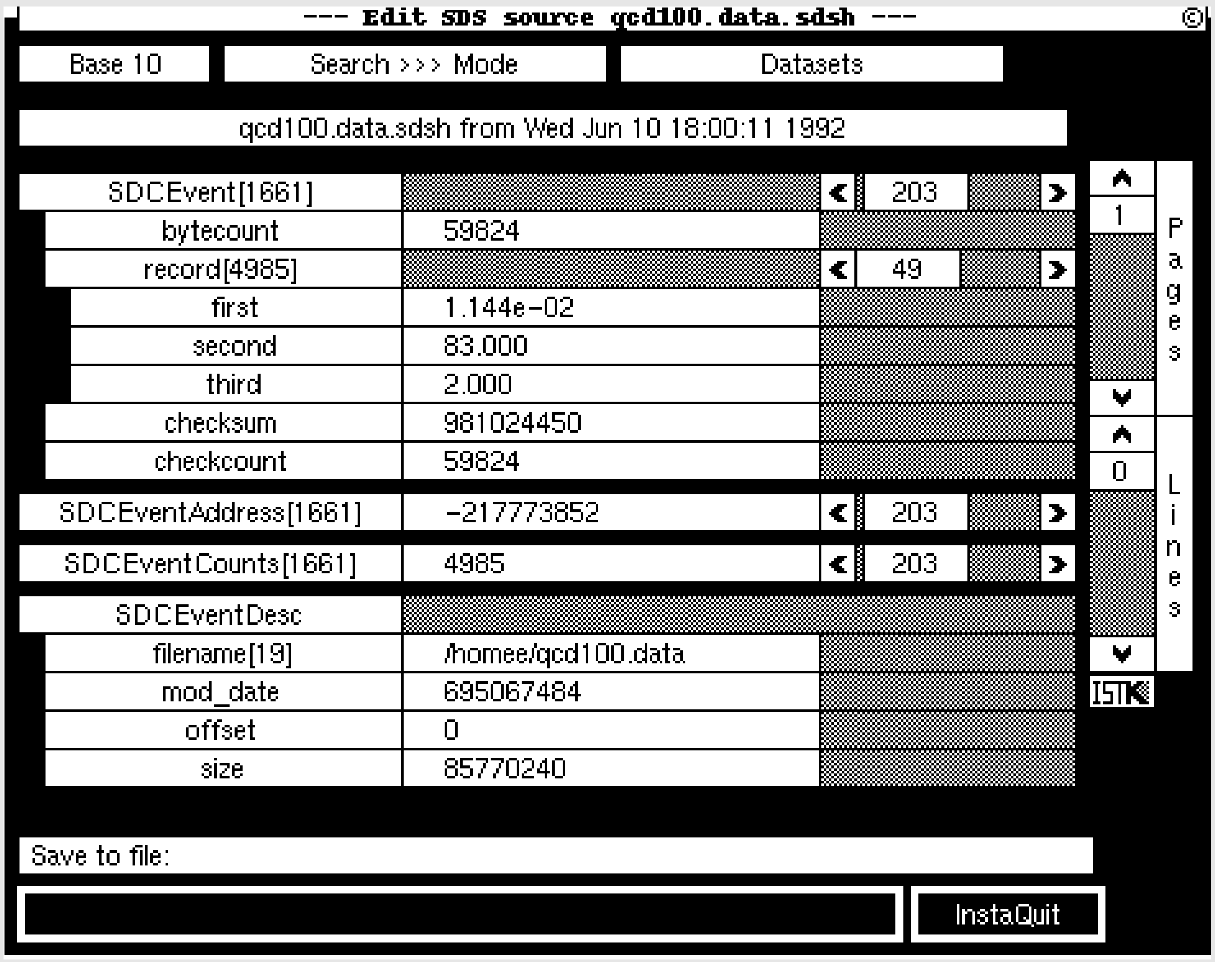
SDS also aids data transfer between programs within a computer, using names and internalized pointers to sort many of the data coherency problems of shared memory that exist in compiled languages.

There are tools to aid in constructing an SDS description. Once the SDS exists, data can be examined, converted as binary data between various processor architectures, or moved to various places where it can live on file, in shared memory, or in network messages. Programs currently exist to convert the data to another format, edit the data, copy the data, delete the data, find out what architecture it was originally formatted for, etc.

Using SDS to describe data also allows for potential upgrade. Should a system become available that is more appropriate than SDS, SDS will allow for transition to that system. Without the use of SDS (or another data-standard), the transfer of information into a new environment will be difficult.

SDS - Data Transfer and Memory Allocation

SDS assumes that, optimally, an entire operation will be automatically run by computers and, in designing SDS, a decision was made to build a structure optimized for the systems with the highest rates of performance rather than building a system optimized for the slowest systems (people). In SDS, data remains in two parts: the basic data objects, and



qcd100.data.sdsh from Wed Jun 10 18:00:11 1992

SDCEvent[1661]		203
bytecount	59824	
record[4985]		49
first	1.144e-02	
second	83.000	
third	2.000	
checksum	981024450	
checkcount	59824	
SDCEventAddress[1661]	-217773852	203
SDCEventCounts[1661]	4985	203
SDCEventDesc		
filename[19]	/homee/qcd100.data	
mod_date	695067484	
offset	0	
size	85770240	

Save to file:

Here, the sid editor is attached to a large (> 80 Megabyte) experimental data file. The header contains all type and size information, including the three lower objects SD-CEventAddress, SDCEventCounts, and SDSEventDesc. These objects contain file information, event pointers, and sizes extracted from the body of the data by a previous scan.

The data itself is mapped from disks. Random access to each event is efficient. Editor customizing is done with a resource file. In this example, the three top objects are ganged together so that accessing SDCEvent [203] will automatically access the lower objects (also [203]). The number of records within each event has been read from the datafile and written explicitly in the SDCEventCounts object. In general, each event is of a different length.

the data structure description. This two part division assures that if data searching is needed it can be done as efficiently as by any interpreted language, and at times if and when data searching is not needed, data access is optimized for use by compiled languages. Because SDS stores data in a format appropriate for high-speed manipulation, the system would be useless without interface tools for the interaction between people and data. To this end, editors and viewers have been developed and are provided with the SDS package.

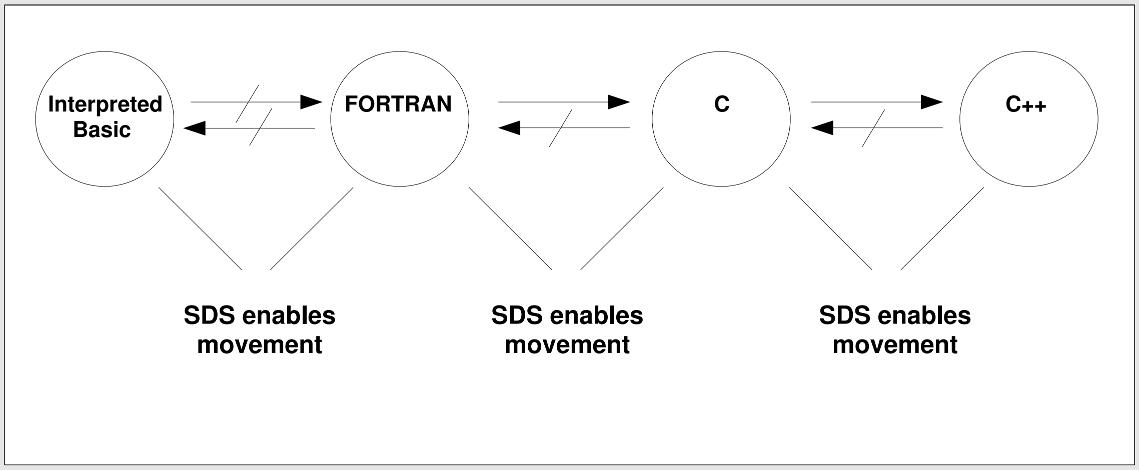
Sharing information between processors and between computer languages is a general problem in the computer world. Different computer systems interpreted languages embed data structuring information in their data, information that cannot easily be read by compiled languages. Some data structures are more sophisticated and complex than others. One of the major problems is that every computer stores its data structures differently.

In SDS, the meta-data that exists in a compiled language only during the compilation period remains as information directly associated with the data. Unlike in an interpreted language, however, no descriptive information is embedded in the actual data. And unlike in a database management system (which must also contain this meta-data), the full description may be moved with small data sets.

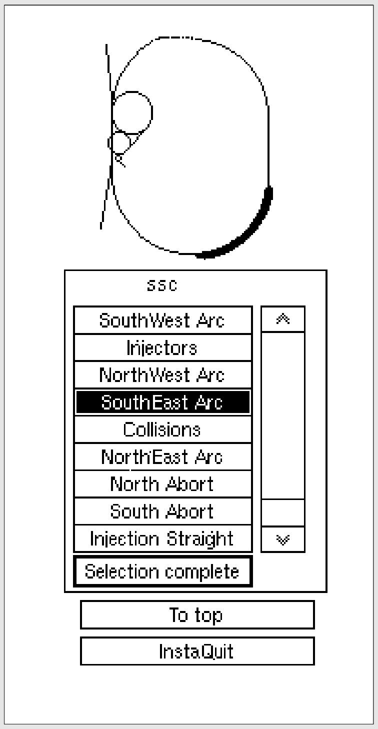
SDS formalizes memory allocation so that the programmer doesn't have to remember how much memory has been allocated. In order to keep track of memory allocated, part of the structure of the SDS header describes the size information of all of its data objects. The SDS description also contains the original memory allocation requests; it enables the programmer to wait until run-time to allocate memory, and the SDS description can be written either at compile time or at run-time.

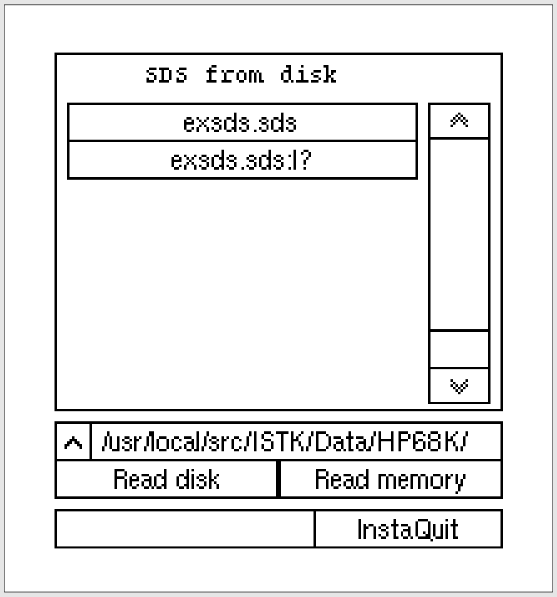
SDS attempts to use the best facets of the memory allocation techniques of interpreted and compiled languages in the right order (keeping the speed of C and the informative power of the interpreted language, but not in a way that slows down C) and then attempts to make it possible to move these pieces of allocated memory from process to process and across networks.

SDS puts data onto the network in the format it has been programmed in and provides enough information about the structures, sizing, memory allocation, etc. so that any computer can translate the data into a readable format. By presenting the data in its own format, SDS avoids the problem of unnecessary translation.



SDS places a layer on top of the data that defines the structure for use by all systems and languages. It allows data from more sophisticated structures to move into less sophisticated structures by providing a bridge over which the structure can travel. SDS enables data transfer in any direction but cannot alter the conceptual structure of a language.





Two "Navagation" Aids

The illustration on the left shows a graphical/textual selector for SSC accelerators. The illustration above is a dataset selector which displays file directories and SDS datasets.

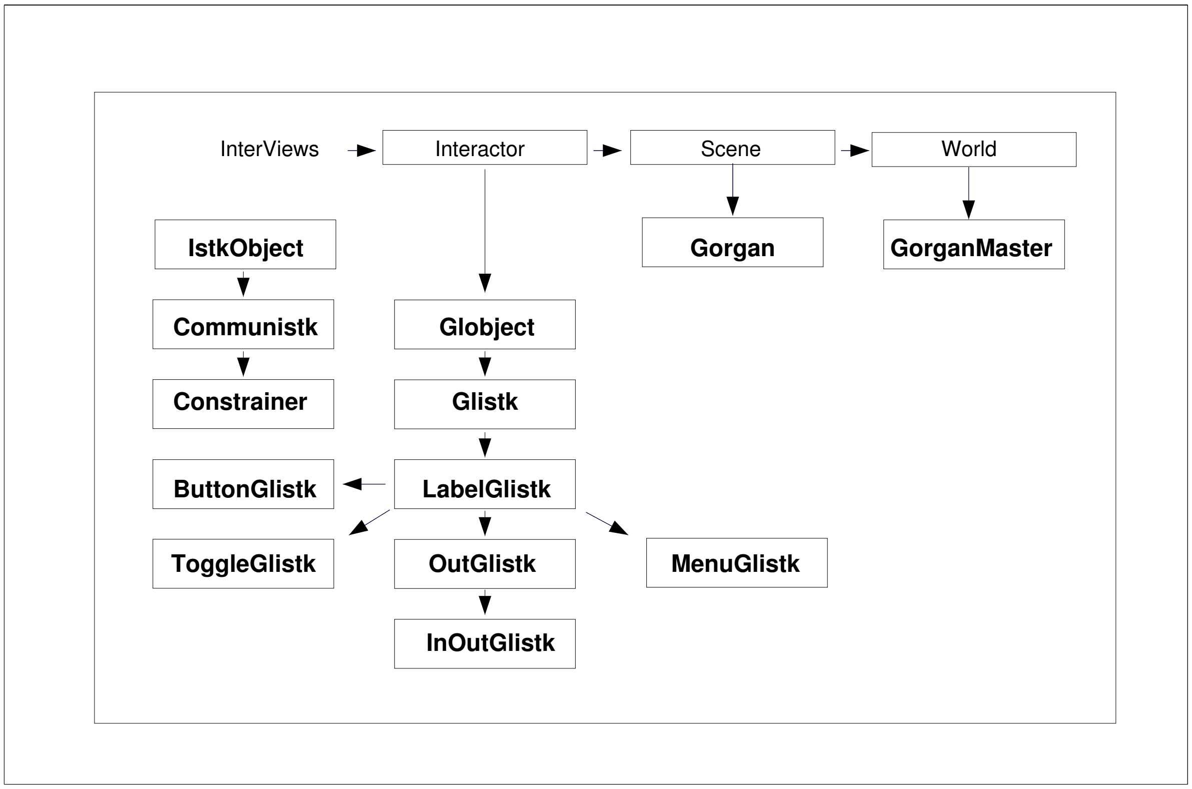
GLISTK Track 1

The Graphic Library for the Integrated Scientific Tool Kit

GLISTK is a C++ library of graphic tools developed to enable rapid creation of scientific computer interfaces for unobtrusive data-viewing and data-manipulation. Because GLISTK is designed specifically for experimentalists, it shrinks the time between the conception and execution of working, connected scientific interfaces.

A good interface is a communication feedback loop between a person instructing the computer what to do and the computer telling the person what it has done. GLISTK provides subtle but important features to strengthen and clarify the feedback connections between the person and the computer through its "no-frills" aesthetic display, through highlighting, through mouse button simplification, and through the lack of icons labeling GLISTK objects.

The Glistk Family Tree



A primary difference between GLISTK and other interface-building tool kits lies in GLISTK's ability to communicate between graphic objects and its ability to extend that communication into the outside world. By having a non-graphic object as its base, a GLISTK interface can be operated by other processes as well as by people. Communication is facilitated by the ISTK communication class, the Communistk. Communistks make the interior connections between GLISTK objects as well as provide linkage to the outside world via the Glish executive.

All of GLISTK's features have been designed with "data purity" in mind. GLISTK objects provide a view of a Communistk's data focus and provide the means for altering data, but they do not affect the data by its display.

GLISTK objects are basic building blocks from which other, more intricate and specific interface tools can be easily derived. While decisions have been made about the look and function of every object in the tool kit, almost every object method is "virtual" so that those functions and appearances can be changed.

The Gloss library is an extension of GLISTK. Gloss objects are derived from GLISTK general objects to form more complex objects such as: Plistk, a graphic plotter; DisLis, which displays the ISTK list class; and Glider, which combines decrement and increment ButtonGlistks with an InOutGlistk. Gloss is a collaborative project and people are encouraged to add their own derived objects to the library.

Credits and References

The Integrated Scientific Tool Kit (ISTK) was developed directly out of the problems associated with running particle accelerators. Development began in the early 1980's by Christopher Saltmarsh at the CERN Super Proton Synchotron in Geneva. SDS, developed by Saltmarsh, was used at the E778 FermiLab in which non-linearities were added to the Tevetron magnetic field to simulate potential S.S.C. control problems. Other ISTK components used at this experiment were Glish written by Vern Paxson at the Superconducting Super Collider Central Design Group, and extended to its present version at LBL, and GLISTK, the ISTK graphic library, developed primarily by Matthew Kane. SDS (and the other ISTK components) are under continuing development and are currently being used at various sites internationally. Continuing ISTK development includes work by Matthew Fryer, Andrea Young, documentation by Ericka Lutz, and contract work from Soam Acharya of Cornell University and Sriram Chittathoor of the University of Connecticut. There have been considerable contributions from user groups such as the S.S.C. Magnet Test Division. The current list of ISTK collaborators (see the mail exploder on largo.lbl.gov) includes people from: the Superconducting Super Collider Accelerator System and Magnet Divisions; Lawrence Berkeley Laboratory; FermiLab; Cornell University; Lawrence Livermore Laboratory; Texas University; and the Texas Accelerator Center.

ISTK Implementation

-The base computational environment for ISTK is Unix.(*), although the SDS component is ported to VMS and PC systems.

-The current SDS implementation is in C with a C++ wrapper and a Fortran binding.

- Glish functions are available from C++ and C.
- GLISTK is presently based on the InterViews graphics package.

ISTK Information and Documentation

All code and documentation is available via anonymous ftp from largo.lbl.gov. The main postscript manuals are:

Lutz, Ericka, and Matthew Kane. The GLISTK Manual . March, 1991

Lutz, Ericka.The Integrated Scientific Tool Kit:, An Overview of the ISTK System and its Components. Superconducting Super Collider publication S.S.C./man 012. July, 1991

Lutz, Ericka. The SDS Document, A Conceptual Basis Towards Understanding the Self-describing Data Standard. December, 1991

Paxson, Vern. The Reference Manual for the Glish Sequencing Language. (vern@ee.lbl.gov)

Paxson, Vern and Chris Saltmarsh. Glish: A User-Level Software Bus for Loosley-Coupled Distributed Systems. Extended Abstract July 21, 1992. Submitted to the Winter 1992 USENIX Conference.

Man pages,notes and technical documents are also there. Enquiries may be addressed to

salty@largo.lbl.gov
matt@largo.lbl.gov

* UNIX is a registered trademark of UNIX System Laboratories, Inc.