The SDS Document
A Conceptual Basis Towards Understanding the Self-Describing Data Standard

Ericka Lutz and Chris Saltmarsh
December 1, 1991

TABLE OF CONTENTS

THIS IS SDS
SDS, the Self-Describing Data Standard, is a component of the Integrated Scientific Tool Kit (ISTK), a computer software tool and component kit developed for use in large experimental control systems. The SDS component of ISTK facilitates data flow: the movement of data between computers and between applications.

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.

THIS IS A TEACHING DOCUMENT
To understand why SDS has been developed and how it operates so that it can be used effectively, it is necessary to understand the overall concepts of data management on a deeper level. This document has been prepared for that purpose. This text has not been designed to be used in a computer science course; it is a simplified and generalized explanation that provides the conceptual framework within which SDS can be understood.

SDS is a system that can be used at various levels for progressively more complex tasks. For simple data description and transfer, SDS can be used without much explanation. However, for more complex processes, a stronger background is useful. To understand what SDS can and cannot do, any users or potential users should read this document before using SDS. (For very experienced programmers, much of the material in this document will be review and can be skipped at your discretion).

In this document, we discuss how SDS has been developed in response to the experimenter's problems and to problems in existing techniques, and we review some of the fundamentals of low-level computer function; memory allocation and data structuring in complete computer environments, within the compiled languages of FORTRAN, C, and C++, and within the interpreted language of Basic.

We also review higher-order naming functions and memory allocation in process and shared memory, SDS general organizational structure objects (SDSListks), persistent objects, and object-oriented programming (with a discussion of two ISTK components: Communistks and GLISH).

WHAT SDS DOES
In fundamental terms, SDS is a description of the structure (byte ordering, name, floating point coding, etc.) of a selected quantity of data (a dataset). SDS binds a descriptive information header to data; this layer of information about the data helps facilitate data manipulation and data transfer between processors of diverse architecture and between various applications within a computer. SDS does not control or determine data formats; the person generating the data determines the appropriate format.

By using the SDS description, every application in the system, whatever its "world view"[1], has the ability to utilize the same data. 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.

[1]Every computer language has its own "world view" towards viewing data and there are often dramatic differences between the world views of different programming languages. This "world view" is often one of the reasons to choose to work with a particular language.

EXPERIMENTAL SYSTEMS AND DATA FLOW

There have been a number of commercial (often business-oriented) systems designed to address the problems that SDS addresses: how to manage the flow of data within and between computers. These commercial systems are not always appropriate for experimenters working with large experimental control systems; their emphasis is geared to the commercial world which has different data management requirements. Many commercial systems too closely couple the data structuring problems with the applications.

During the operations of complex, experimental equipment, vast quantities of raw data are collected by numerous widely-distributed computers and instruments. These megabytes of incoming data must be clearly understood as well as be accessible for short and long term analysis. Keeping track of this data is one of the experimental programmer's biggest concerns, especially because it is difficult to predict exact technical specifications or to predict what problems will arise that will require data restructuring.

Because it is difficult to ascertain which data are pertinent before some analysis has occurred, much of the incoming data must be stored until the preliminary processing is complete.[2] It cannot be pre-determined who will need access to particular pieces of data or what type of computer and applications they will use for analysis.

Whatever is in charge of managing these quantities of incoming data must clearly understand what information has been gathered, must identify its origins, and must be able to transfer that information from processor to processor for analysis.

Dealing with Data Flow Management

There are several existing ways to approach the problems of data flow management. In an experimental situation, people have generally used existing interpreted or compiled computer languages, changing existing tools to fit the job rather than custom creating tools for the situation. This isn't always effective; it is always better to have the "right tool for the job" than to make do with tools designed for other purposes. More recently, complete environment and object-oriented languages have been used. These integrated systems, while very effective in certain circumstances, are also not necessarily appropriate for the experimenter.

Integrated Systems - Ideal <u>If</u> They Could Be Universal

For keeping control of data moving from process to process and across processors, a single system seems desirable.[3]

> **"The advantage is constancy, which shows up throughout the system. The whole integrated system is organized by a single principal, so a user need understand only this principal to explore any part of Smalltalk-80."[4]**

In complete computer environments, the communication system works well, programming is easy, the debugging tools are effective, task-appropriate tools can be built and a graphic interface is included.[5]

---

[2] It is sometimes not possible to store all the incoming data.

[3] One example of a good single system is Smalltalk-80, which provides both a complete environment and an object-oriented language.

[4] Cox, Brad J. and Andrew Novobiliski. *Object-Oriented Programming : an evolutionary approach*, 2nd Ed., Reading, Mass.: Addison-Wesley, 1991, pg. 33

[5] Integrated systems typically don't easily lend themselves to sequencing but even that can be done.

If using a single system to facilitate data flow were feasible, SDS (and all the other Integrated Scientific Tool Kit components) would not be needed, as an integrated system provides a more coherent interface (and easier memory allocation, data structuring, data transfer) than the general purpose interpreted and compiled languages can provide.

However, integrated systems have their drawbacks. They are not yet efficient, perhaps too user-friendly (people can use them without a strong conceptual background) and costly. The reality is that people will continue to use other languages in various manufacturer's computers, all of which have slightly different ways of doing things, and will continue to need to share data generated in various languages and systems with each other. While the number of languages and processors used can be limited, it is not realistic to imagine that everything can be standardized.[6]

Unfortunately, using a single integrated system is not an option at this time. Given that processors of different types and languages of different types will continue to be used, the Smalltalk-80 type of complete environment can be effectively eliminated for overall control; it can only be used as a component.

Interpreted and Compiled Languages

Broadly speaking, there are two kinds of computer languages: interpreted languages and compiled languages. We will be concentrating on BASIC (interpreted) and C (compiled).
(NOTE: For the purposes of this demonstration we use an extreme approach. In reality, the differences between various compiled and interpreted languages are less stark; as with most things, they exist in shades of gray.)

[6]It is good to remember the ADA story. When the Department of Defense determined that 420 programming languages were being used in their facilities, they decided to develop ADA as a universal system that could handle every possible application. The result: the Department of Defense has 421 languages to contend with.

Interpreted and compiled languages translate into machine code in different ways:

Interpreters	Compilers
Time 1 Read a line Check for errors Convert to machine language Execute line	*Time 1* Read entire file Convert to machine language Optimize Try to execute compiled program Check for errors
Time 2 Read a line Check for errors Convert to machine language Execute line	*Time 2* Execute compiled program

Differences Between Compilers and Interpreters: Interpreters convert each program line to machine language each time the program runs. Compilers convert to machine language only once and then on subsequent times run the already converted commands.[7]

In a world of infinitely fast computers, interpreted languages alone might be sufficient for experimenters and SDS (or compiled languages and integrated computer environments) might not be necessary. An interpreted language provides quick feedback (informing the programmer of errors): all the data in its memory are named, easy to locate and easy to edit; everything can be saved; and prototype building and creation of other small programs is quick and effective.

At this point in time though, interpreted languages tend not to be fast enough for the experimenter's needs. In an interpreted language, data are searched sequentially and systematically every time the program is run. There are ways to speed up an interpreted language's performance and give it more efficient search methods[8] but, whether an interpreted language is used in its most basic form or uses accelerated search methods, information about the data is embedded into the data themselves. This embedded information becomes a problem when the data are transferred between processors and from interpreted languages to compiled languages.[9] To transfer data created by an interpreted language into a compiled language program requires moving the data through the separate processes of reformatting it, changing it, and filtering it. Typically, the data must be copied. These extra steps all slow the data transfer process.

Compiled languages are generally faster than interpreted languages because the compilation process has a better opportunity (the optimization process) to promote efficiency. This same advantage can also be a disadvantage: in order to make any alterations, however small, in a compiled language program, elements of the program must be re-compiled and linked. Another serious drawback with compiled languages is that information about the data (meta-data) is lost after compilation and it becomes difficult to reference what is contained in a block of data without searching through the code. Compiled languages, however, provide more data-structuring flexibility than interpreted languages do and, in an experimental situation where data-structuring flexibility and speed are vital, an interpreted language is at a disadvantage to the more sophisticated compiled languages.

---

[7]Adapted with changes from: Wilson, Stephen. *Using Computers to Create Art.* Englewood Cliffs, N.J.: Prentice-Hall, 1986 pg. 161

[8]Programs can be speeded up by a number of methods but interpreted languages are intrinsically slow.

[9]Compiled languages make assumptions that there is no embedded description of the data in the code.

The SDS Approach
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 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.

WHY SDS?
SDS has not been designed to replace integrated database management systems. SDS combines data management techniques of interpreted and compiled languages and provides management control when data are being manipulated outside the database management system. The comparison between compiled languages and interpreted languages was not meant to argue the benefit of using of one system over another but, rather, to point out that neither system is perfect.

Data Transfer / Memory Allocation / Data Structure
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. Because computers store their data in varying formats, data transferred between systems often can't be read without a time-consuming translation process. Often, the "world views" (the idiom that a language uses to describe data) are not compatible and the languages have different approaches to looking at data (making problematic, for instance, the transfer of complex data structures generated in a compiled language to a system using an interpreted language).

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. To understand how SDS binds these components, we need to look at the differences in data transfer and data structuring techniques in different systems/languages. We also need to look at memory allocation techniques in compiled and interpreted languages to clarify how SDS operates.

MEMORY ALLOCATION IN COMPILED LANGUAGES - C
In an experimental situation where it is impossible to determine how much memory is needed until objects have been created, it is difficult to pre-allocate memory. In the higher level compiled languages, memory allocation is formalized to allow the existence of dynamic objects of complex different data structures whose memory allocation is determined at run-time.

The following example is a demonstration of both automatic and run-time allocation in C language.

Sample Fragment of C code[10]

```
int a;
int *b;
a = 3
b=(int *)malloc (size of (int));
*b = 4
printf ("%d\n",a);
```

Automatic Allocation

In the sample code above, when the compiler reads the command, **int a;**, the compiler allocates four bytes of memory in the stack[11] as the integer **a**.[12] The compiler keeps a pointer that instructs the computer: should there be a reference in the program to **a**, point to this area of memory.

The command **a=3** writes the number **3** into the area of memory (the four bytes) that has been allocated in the stack.

The command **printf("%d\n",a);** does, indeed, refer back to **a**. It tells the computer to print out the integer **a**. When the computer receives this command, the computer then goes to the determined place in memory (those four bytes in the stack), finds the value of **a** (which we have already determined to be **3**), and prints it.

Run-Time Allocation

Something different happens when the compiler reads the command **int*b;**. Because * means address, the compiler understands that **b** is not an integer itself, it is a pointer to an integer. Since a pointer takes up four bytes of space just as an integer does, the compiler allocates four bytes of space in the stack. This pointer now points at some undetermined place in memory.

The command **b=(int *)malloc (size of (int));** allocates memory for **int b** at run-time (as opposed to automatic allocation at compile time as is done with **a**). When the program actually runs, the computer will allocate enough space for an integer. The computer finds another four bytes, in a completely different part of computer memory (not on the stack), and determines it as the area of memory that **int*b** points to. The address of the new, allocated space is now written into the pointer **int*b.** Once the memory is allocated, ***b=4** writes the value of **4** into the new, allocated space.

In our example, using run-time allocation seems silly because it would be very easy to simply allocate space for the integer **b**. Imagine, however, an experimental situation where a piece of hardware is emitting numbers and measurements. Because the size and even the type of the numbers may be unclear until run-time, having the flexibility of being allowed to allocate the memory space as needed is a decided advantage.

[10] A final line in this code, much much later, would be: **free (b);** which frees the memory allocated for **b** and puts it back for future use.

[11] Stack - according to legend, this data structure was inspired by spring-loaded stacks of cafeteria trays. A stack is a buffer - things are pressed down into the stack and they only come out of it in the reverse order they were in pressed in - last in = first out. Program stacks are used at run-time. When a program is run, addresses, variables, all the things that a program needs to know about are inserted into the stack.

[12] An integer takes up four bytes of space in most computers.

Common Memory Allocation Errors in C

A common mistake in the example above is to forget that space must be allocated before it is used, to use the pointer without saying what it is going to point to, and to accidentally write the value (in this case, **4**), rather than the address of the value, into the pointer itself.

In the more sophisticated compiled languages, the programmer is allowed to allocate memory as needed, but must take responsibility herself for memory allocation. There are several steps involved but the basic principal remains the same: space must be allocated before it is used and still be allocated when it is used.

The chart below describes a common allocation error.

WRONG Example of Allocation in C

char*;	It is a char*.
getname();	It is a function called getname.
{	Start body of code.
char name [20];	Find space for 20 characters.
strcpy (name,“hello”);	Copy into that space the literal string “hello”.
return name;	Return with the pointer to that piece of space.
}	End code.

Unfortunately, this code MIGHT work (for a while). However, it is illegal. Why? It is not enough to simply name the object and set memory aside for it in the stack.

When the compiler reads the sample code **char name [20];**, it knows that the programmer has requested some memory. It allocates it in the stack.

At the command **strcpy (name,“hello”);**, the computer writes “hello” onto that piece of memory. This memory is not reserved after the function returns. The code will only work until the compiler happens to use the memory for something else (as it is completely allowed to do). At that point, the memory will be overwritten, causing later problems.

The command **printf(“%s”,name);**, if given within the function, will print out the name: “**hello**” will appear. The scope of the object, however, exists only within the function it has been written in and if this command is referred to within another function, the computer will perform the command but important information inside that piece of memory may have been overwritten (and printing that data may be fatal to the program[13]).

[13]For example: in C, a character string is terminated with a 0. Printing the string searches for that 0. If it has been overwritten, the computer will continue to print and may cause a fatal error.

The chart below describes a correct example of a memory allocation code fragment.

RIGHT Example of Allocation in C

char*;	It is a char*.
getname();	It is a function called getname.
{	Start body of code.
char*name;	I need a temporary character pointer.
if(name=(char*)malloc(20))==NULL) exit(0);	“If” statement to trap malloc.
strcpy (name,“hello”);	Copy the literal string “hello into it.
return name;	Return a pointer to the explicit piece of memory where the name is.
}	End code.

A Note About Memory Allocation in FORTRAN

In FORTRAN, a base-level compiled language, the programmer cannot allocate memory at run-time. To use FORTRAN, the programmer must allocate as much memory as she thinks will be needed. FORTRAN reads only as much data as memory has been allocated and no more, so usually, the maximum amount of memory is requested. This has some disadvantages: every program tends to become huge, and sometimes the programmer runs out of memory.

Lost Information and Overwrites in Compiled Languages[14]

In compiled languages, structuring information about the data (meta-data) is often stripped to make the program tighter after the compilation process. The result is that it is difficult to reference what is contained in the block of data without searching through the code.[15]

[14] In interpreted languages, while all the data structuring information is available at run time, an unfortunate reality is that it remains either inaccessible or specific to the interpreter that created the data. In actuality, programmers using interpreters must write data structure information in two or more places.

[15] In FORTRAN, for instance, only a small portion of data structuring information remains after compilation. C retains slightly more information.

Referring back to our sample fragment of C code:

Sample Fragment of C code

```
int a;
int *b;
a = 3
b=(int *)malloc (size of (int));
*b = 4
printf ("%d\n",a);
```

Once this sample code is compiled and the program is run, the meta-data about **3** (that the programmer named it **a** and that it is an integer) is never used; though it remains around (but not in!) the code unless it is stripped out, it is only used for debugging.[16] Once the original information about the data structure is stripped away it is gone forever, leaving only a machine code pointer pointing to a memory location.

Stripping the code of meta-data can lead to data overwrites: in some compiled languages, pointer space allocated for an integer can be used to store other data structures. For instance, if the programmer specifies space for a floating point number but then writes in a double, the compiled language will do it but will overwrite her memory. This is less of a problem in interpreted languages.

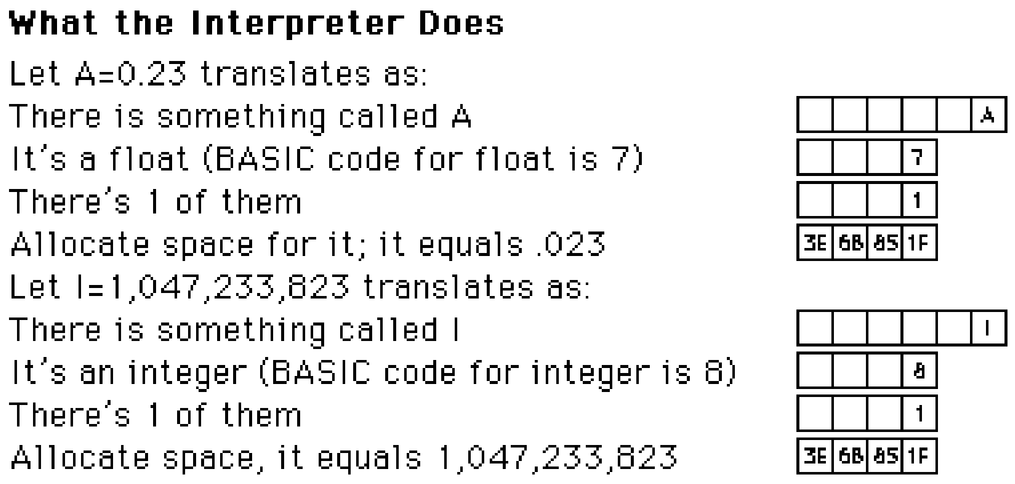
MEMORY ALLOCATION IN INTERPRETED LANGUAGES

Interpreted languages work differently than compiled languages. In an interpreted language, all memory allocation is done at run-time and is automatic. Code is searched through sequentially every time the program is run. As it searches, the interpreted language checks to see if a data object that is referred to exists in memory. If the data object is nonexistent, the interpreted language will allocate memory for it.[17]

Sample Fragment of BASIC code

```
Let  A=0.23
Let  I=1,047,233,823
print  A,I
```

[16]If something goes wrong, the debugger will use the symbols to help find the problem.
[17]The fact that code is syntactically correct doesn't mean that it is going to work. Because allocation in an interpreter is all done at run time, within a program there might be a sub-routine which very seldom gets attached. The program may hit the sub-routine and suddenly find that it is out of memory.


What the Interpreter Does

Let A=0.23 translates as:
There is something called A
It's a float (BASIC code for float is 7)
There's 1 of them
Allocate space for it; it equals .023
Let I=1,047,233,823 translates as:
There is something called I
It's an integer (BASIC code for integer is 8)
There's 1 of them
Allocate space, it equals 1,047,233,823

Demonstration of a Crude Search Mechanism
When the interpreter reaches the command, **print A,I**, it begins to search for **A** from the top of the data heap. Recognizing the first item as a float, specifically, the float **A**, it moves down to the next line to find its value, and prints out **1.792**.

After having found and printed **A**, the interpreter returns to the top of the data heap to search for **I**. The first thing it finds is the float **A**. As that is not the integer **I**, the interpreter continues to the beginning of the next item, skipping the rest of the description of **A**. It it able to do this because it knows that a float plus its description needs a particular number of storage spaces in memory. It now jumps down that many spaces and checks the next object. In this case, the second item is, indeed, the integer **I**. The interpreter moves down to the next space to find the value of the integer **I.** Finding that value, **32** it is now able to print it out.

This is a crude process. Many ways exist to speed up it up (binary searches, etc.), but the important thing to remember is that there is a lot of meta-data (that it is a float, that its name is **A**), deeply embedded in the data itself. In an interpreted language program, this meta-data <u>must</u> exist, unlike the symbol tables of a compiled language program which can be stripped away. In our example, the meta-data about the actual numbers **3** and **4** utilizes more memory than the numbers themselves.

SDS' MEMORY ALLOCATION TECHNIQUES/SYSTEMS

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.

One philosophy behind SDS is the belief that all the techniques for memory allocation are fine, but that they have their limitations. It is very useful to have naming, structure, size and location information (not just in symbol tables, but up front, as in an interpreted language). However, the programmer can do without extraneous information being presented every time she looks at her data.[18]

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.

DATA TRANSFER BETWEEN PROCESSORS - LIFE BEFORE SDS

Sharing information between processors and between computer languages is a general problem in the computer world. Different computer systems provide various models for memory allocation. 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.

Data Structures

There is a short list of basic types of data structures (integers, floats, doubles, char strings, etc.). In FORTRAN and in most BASIC type languages, all data structures must fall within those parameters. In more sophisticated languages, (C, C++ etc.), those structures can be more complex.

```
int  value  [256];
int  min_value;
int  max_value;
```

This sample code outlines an array of 256 integers and a minimum and maximum value associated by the naming but not explicitly related to the array. A better way to associate these values would be to define a data structure:

```
struct  val;
{
   int  value  [256];
   int  min_value;
```

[18] Accessing this information about the data can slow the program down. When this information is not needed, the data can be accessed rapidly. Reminder: in SDS, the data remain in two parts: the basic data objects, and 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.

```
                int max_value;
            }
```

This structure defines a new kind of object, called **val,** composed of an array of 256 integers, another integer called the minimum, and another integer called the maximum. These values are inextricably, explicitly mingled, not just by the naming as in the example before, but by the data structure itself. Once the generic structure exists, the programmer can request one and can define its **val min** and **val max**, determine the contents of the value array, etc.

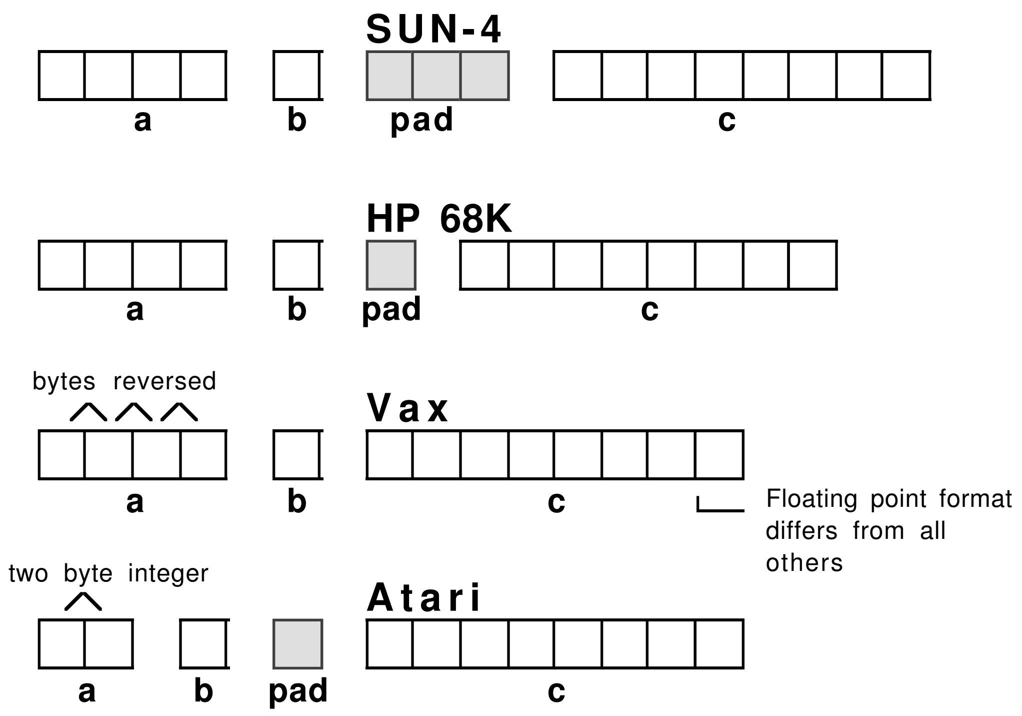
Varying Data Structure Storages in Compiled Languages[19]

When data is transferred between compiled languages, the data structuring information is stripped after compilation and the data is transferred in binary form. These binary numbers can be interpreted in any way. The computer has no way to know how the data was formatted on the computer it was created on, and therefore has no way to differentiate pad bytes from significant bytes.

[19]The problem with varying data structures is not limited to compilers; data created with an interpreted language on one machine, when given to the same interpreted language on another machine, will not necessarily be understandable.

The graphic below illustrates some varying data structure storages for the following data structure:

```
struct fred
{
  int a;
  char b;
  double c;
}
```



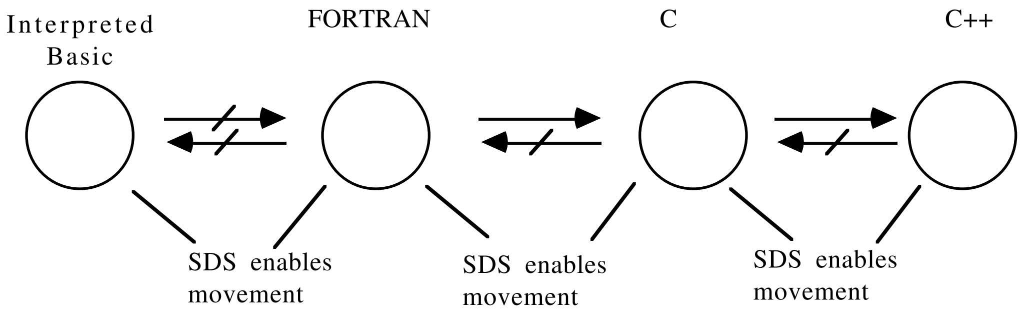
Dealing with Data Transfer - External Data Representation
Data transfer between processors is not a new problem in the computer world and various ways have been invented to deal with it. One example: a division of Sun MicroSystems promotes a technique called "External Data Representation." When transferring information between two unknown computers, it automatically translates data into a specific format as it is being put onto the link. Any computer receiving the information will know, for example, that there are no pad bytes in the data, etc.

This techniques works. Its major drawback is that each process automatically translates its data into this format even if they are going to the same type of processor (an extraneous third step), and the data structure must be already known by both ends of the system; no meta-data exists with the data itself.

THE SDS APPROACH TO DATA TRANSFER

SDS has been designed with the belief that a data flow facilitation system should manage data as it has been written and not put structure or memory constraints on the programmer. SDS allows the programmer to determine what data structure model she wants to use for the data. The SDS header then simply describes that model.

SDS puts data onto the network in the format it has been programmed in but, unlike with the "External Data Representation" technique, 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. Data cannot be directly transferred between Basic and FORTRAN. Data can be transferred from FORTRAN to C, but it can be difficult to transfer data from C to FORTRAN[20]. Data can be transferred from C to C++ but because of differing data structure, transfer from C++ to C is difficult. SDS enables data transfer in any direction but cannot alter the conceptual structure of a language - data transfer of complex data structures to less sophisticated is difficult even with the SDS description.

SDS presents a lump of binary data which can be mapped efficiently, while allowing the data structuring information to remain accessible to the programmers by placing it outside the data. Any piece of published or transferred data keeps the descriptive information with it. Programmers seeking access to the data are presented with mechanisms which inform them what is inside the block of data.

For our data structure example,

```
struct  val;
{
   int  value  [256];
   int  min_value;
   int  max_value;
}
```

the SDS description would explicitly state that this structure has an array of 256 integers followed by an integer called the min_value, followed by an integer called the max_value. It would specify that its name is **val** and that this data came from a computer with, for instance, a “R.I.S.C.” padding strategy. By having all this information, a computer with, for instance, a “Motorola 68000” or a “Vax” padding strategy can understand how to translate the data to a format it can understand.

TRANSFER WITHIN A SYSTEM

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. The following discussion will review memory allocation techniques within process and shared memory in order to clarify how SDS operates in this area.

ALLOCATION WITHIN PROCESS AND SHARED MEMORY

[20]To transfer a repetitive structure from C to FORTRAN without SDS, FORTRAN must allocate the memory. Typically in this form of data transfer, the data will have to be copied.

Within a computer's process memory, a program with all its instructions and data are stored in a piece of memory. Should the program need more memory, it can ask the computer to allocate more space. This additional allocation happens in one of three ways depending of the type of computer being used:

In computers which don't use a memory manager (real time systems, Apple II,[21] etc.), the RAM is directly accessed. As a result, the memory is not protected and any process can affect it.

In computers with a memory manager, the program requests memory from the memory manager. When the memory manager accesses the memory, it gives the program a pointer to that memory. That pointer may be an indirection; it may not map directly to the actual memory location but, because the memory manager is policing the memory access, the actual location of the memory is irrelevant to the program. Use of the memory manager also insures that the memory allocated will be reserved for the program and isolates the program so that any wrongful memory access will not affect any of the other programs.

In computers using a virtual memory system as well as a memory manager, the program is fooled into thinking that the computer has more memory capacity than it really does. The virtual memory system does this by swapping bytes from RAM to disk space to RAM as is needed. In this way, an 8 megabyte RAM system will seem to the program as though it has 100 megabytes. Virtual memory systems work because most programs' data access tends to be localized.

Sharing Data Originating in Process Memory
Process memory provides fast data access but doesn't provide the ability to share data with other programs within the computer. Sometimes more than one program within a computer wants to share the same data. This can be done by filing the data generated in process memory on disk and allowing the programs interested in that data to make a copy of it for their own process memory. However, this can be wasteful and time consuming, many copies of the same information can use up valuable space, and there may be problems with data coherency; if one of the programs updates the data, it must inform all the other programs that have copies (via the disk) how the data has changed.

Shared Memory
Shared memory allows data access to more than one program within a computer. Technologically, it is immensely more effective than process memory when more than one program is using the data. Accessing the live memory is much faster than accessing files and, because there is only one copy of the information, the "dance" of loading from file, copying, altering, returning to file, etc. does not slow things down and data coherency problems can be more efficiently handled. (Of course, using shared memory is also more dangerous than using process memory; as there is only one copy of the data, mistakes are more difficult to remedy and there may be problems with access control[22]).

How Shared Memory Functions
In the Unix system, the interprocess communications suite (ipcs) handles shared memory. When a program needs a piece of shared memory, it requests a particular size of it. If it wants to increase the size, the computer allocates more memory but (as in realloc()) the new piece of memory is not necessarily attached to the old piece of memory. The new memory may exist in a different location.

Thus, if two programs are connected to a lump of shared memory, one generating data and one reading it, and the program generating data realizes it needs more space, ipcs may allocate that space in a different

[21]In real-time systems, direct memory access provides the advantage of speed; there is no memory manager in the way. The Apple II has no memory manager because it is a simpler machine.

[22]The memory manager has some control of access, and there are ways to put blocks on data, but there are always control flow problems and a piece of data might be accessed by a process before the generating process has finished its input.

place in shared memory. The generating program will continue to write into it. However, the program reading the shared memory data will still be attached to the piece of memory containing the old data and it will stay there until it is instructed to release it and and the connection is broken. Only once it has been instructed to do so can a new connection can be made to the new piece of shared memory. In order to avoid this situation, a program-to-program control message structure is needed.[23]

SDS AND SHARED MEMORY

Because of the intrusion of the memory manager, different systems cannot use the same addresses (absolute pointers) to access data in shared memory. If one program writes a pointer into a data object, that pointer will not point to the same piece of data in another program. In shared memory, the normal practice of leaving pointers to objects is extremely dangerous.[24]

SDS sorts out many of the shared memory data coherency problems of compiled languages that exist because of the fact that absolute pointers can't move between programs. SDS allows data objects to be addressed within shared memory either by name or by guaranteed internalized pointers. (Internalized pointers are pointers that do not point directly to a data address, they are set offset to the data, usually relative to the beginning of the data.) For each program, the SDS code will give the correct pointer to the offset position.

WARNING FOR SDS USERS: When moving SDS datasets from from process memory to shared memory, please be VERY CAREFUL! A common error is to continue to use process memory even AFTER the data has been copied into shared memory.

DATA DESCRIPTION

SDS serves not only as "glue" between the bits and bytes of various systems, languages, and applications, but also operates on a higher level of data description. One of the important SDS data description features is the SDSListk, a type of list.

LISTS AND ARRAYS

A list is a very general organizational data structure, a way of associating data objects. When data objects are associated in a list, they can be easily searched for, reorganized, etc.[25] While many of the things done with lists can also be done using arrays, it is far easier to organize data objects with a list. (Arrays, in fact, are actually a simple type of lists; automatically built in to many computer languages.) While lists require the additional effort of extra code, they are ultimately far more flexible than arrays.

List code associates things in a linear or a multi-dimensional structure. The items in the list exist in memory connected only by list pointers that go from item to item. To add a new name to the list, all that must be done is is to change the pointers, to break and reset the linkages. No duplication of data items is needed and, in the SDS system (SDSListks, the SDS list), there is no manual memory reallocation.[26] A list may be multiply linked and data items can be associated in more than one list at a time. General rules may be specified for all items on the list.

---

[23]ipcs handles the basic tools for this (message passing and semaphores).

[24]...and extremely funny, says Chris.

[25]A common example of list usage: in many compiled language systems, the computer doesn't have any up-front information about the size and memory allocation of its data objects. In order to find how much data are stored, the user must keep requesting data until there is no more. This can be dealt with by producing a list; a directory structure, where pointers are kept to each new piece of data.

[26]There are times when memory must be reallocated because the programmer doesn't know how many of a given type of data object she will need.

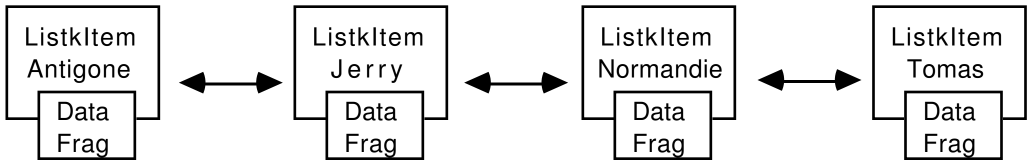
An array, in contrast, is a linear association existing in contiguous space in memory. To add an item to a linear array, a new space in the array must be created and the items already in the array must be shifted in order to insert the new item. In order to associate an item in more than one array at a time, it must be duplicated.

LISTS IN THE C++ IMPLEMENTATION - THE SDSLISTK

Lists are a highly effective organizational tool until the programmer wishes to pass the list outside the program. Because List code relies heavily on pointers, the code becomes unavailable.

The basic ISTK list class is a Listk. SDSListks add persistency to Listks. Whereas basic Listk code requests a pointer to a data object, SDSListk code requires more information about the data object so that it can be identified both internally and externally, allowing lists to be transferred, saved, examined and manipulated by different processes, wherever they might be.

SDSListks list Data Frags: the data being associated by the SDSListk[27]. Around the Data Frags are the ListkItems, objects which serve as descriptive wrappers.[28]



An Alphanumeric List

A ListkItem contains a primary and secondary pointer, the name of the Data Frag, and its identification number. The primary pointer in a ListkItem points to the actual code which gives access to the Data Frag. This code cannot be saved, only a summary of it (the descriptive name)can be saved.[29] The secondary pointer points to data which defines how the code object (Data Frag) will be used. This data can be saved and, once reconstituted from file or memory, these encapsulated data "specs" can be used in a variety of ways.[30]

Data Frags in an SDSListk can be accessed by the name, ID number, or primary pointer. They can be run through (like an array); the user can delete or move them. Users don't create ListkItems directly; they are built when new user pieces are added, inserted, etc, to the list. The ListkItems supervise the Data Frags, make the list connections, and allow Data Frags to be added and deleted.

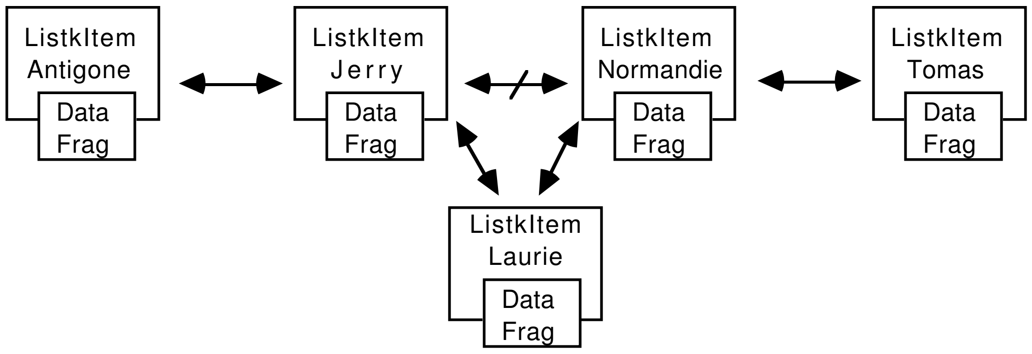
---

[27]Like most list code, Listks cope with pointers to objects, not with the objects themselves. Listks do not allocate memory for Data Frags (data objects) that they list (they can't, they don't know what the Data Frags are). Also, they don't allocate memory for names UNLESS and UNTIL the user explicitly asks for a name heap to be created (in preparation for the SDSListk).

[28]In an array, ListkItems are not necessary, they are contained in a rule that the Data Frags will be placed in contiguous spaces in memory.

[29]If this were a complete Smalltalk environment, the code would be saved as well.

[30]For instance, the user may want a textual description of a picture. She would use the data provided in the secondary pointer and create a different, textual object instead of the original visual object created and described by the original primary pointer.



Adding an Item to An Alphanumeric List

To add a new Data Frag to an SDSListk, the user presents the new Data Frag and requests that it be added to the list.

Persistent SDSListks

SDSListks are persistent, eliminating the problem of transferring pointers from one running program to another. To save an SDSListk for later use, or use by another program, the information referred to by the secondary pointer is collapsed and two new objects are created. The first item is a concatanated array of driving data about the Data Frags. The driving data can include such things as definitions of coordinates, colors, backgrounds, etc. The second object contains an array of the ID numbers, the name heaps, and the pointers; information about how the Data Frags were used. When these two objects are stored and brought back, all the necessary information is there to reconstitute the SDSListk. The only thing that cannot be done is to store the actual code.

GRAPHIC - SID Picture

HIGHER ORDER NAMING CONVENTIONS

There are additional conventions that can be used to augment the low-level SDS headers. An example of higher naming conventions follows:

When transferring data from an SDS to an SQL database, the SQL command **select * from table** will result in a database output that can be mapped directly into a single level C-structure and then loaded into an SDS dataset with all of its meta-data (row names types, etc.) intact. The command **select * from table where table.value >42**, will result in a subset of the data of the previous command (although the data structure is identical). The low-level SDS header has no way to contain the extra meta-data (the fact that the database output is a subset of the table and not the whole thing). Obviously, this can be problematic.

The SDS tool **db2sds** adds another layer of information, a simple higher naming convention, to the meta-data existent in the SDS header. If and when the "where" clause is present, an additional string data object will appear in the dataset named table.where. For the example above, the additional objects value would read "value > 42". This extra data could subsequently be used, for instance, to label a graph of "table" or to act as the qualifier when sds2db (the reverse operation, transferring data from an SDS to an SQL database) attempts to update the original table in the database.

SDS COMMUNICATION - COMMUNISTKS AND GLISH

The Integrated Scientific Tool Kit (of which SDS is a major component) has two high-order communication features, Communistks and GLISH, which SDS uses to access data objects and handle the control flow between programs/systems.

Communistks are the communication class for the ISTK libraries. A Communistk is a data 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 hopefully 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.

For a successful communication set-up between data objects, several things are required: one or more senders, one or more receivers, and a message body. Communistks are built to take over the mechanics of communication checks and routing for a set of communication types. To this end, they take over responsibility for the message body: the Focus.

The Focus of a Communistk

The Focus of a Communistk is essentially 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[31] to the Focus' value to data objects which have expressed an interest in the Focus. The interested 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.[32] 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 Constraints

The Communistk creator, when creating space for the Communistk's 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[33] and illegal actions will be thrown out by the Communistk.

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 change in the Focus resets which value is being looked at (Monday's, Tuesday's or Wednesday's

[31]The message body may possibly be null; the arrival of a message may be a signal in itself. The first release of Communistks notified everybody whenever ANY change happened to the Focus. Release 1.1 allows more restricted changes to be notified, such as passing warning limits. This code is not fully tested.

[32] An object may, of course, be both a changer of the Focus and be interested if anything else has changed it. In that case, though the data object that changes the Focus is already on the list, the Communistk can be instructed not to notify it. Also, when a number of changes are to be made before response by other objects is required, the Communistk can be set to change the Focus value without notifying the data objects on its CommuList.

[33]By the SetValue () method. The data object that is inputting the value change does not change the value of the Communistk's Focus directly.

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.

GLISH

Communistks only work within a process. GLISH, the ISTK component responsible for sequencing and control flow, facilitates communication between different processes and computers, carrying messages from one process to another via the GLISH executive.[34] For the programmer, the only difference between internal and external messages is that Communistks which will communicate outside their process must be named. For external communication, this Communistk name becomes the GLISH event name.

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, a sequencing language rewritten to its present implementation by Vern Paxson at the Superconducting Super Collider Central Design Group, 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 and documentation by Ericka Lutz, and 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 Information and Documentation

Computer files available via anonymous ftp from largo.lbl.gov

Document lists (including SDS and GLOSS man pages) available on largo.lbl.gov in file: /usr/local/src/ISTK/Doc/index (from anonymous ftp these will appear on pub/ISTK).

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

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-0017. July, 1991

Email to:

istk-counsel@psychosis.ssc.gov
istk-gripe@psychosis.ssc.gov

Suggested Reading and References -

Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language.* Englewood Cliffs, N.J.: Prentice-Hall, 1978

Cox, Brad J. and Andrew Novobiliski. *Object-Oriented Programming : an evolutionary approach*, 2nd Ed. Reading, Mass.: Addison-Wesley, 1991

Wilson, Stephen. *Using Computers to Create Art.* Englewood Cliffs, N.J.: Prentice-Hall, 1986

---

[34]As with Communistks, the sender is not aware of where that message may go - it is the job of the external communication router (the GLISH executive) to do this and to cope with exceptions.