Self-defining Data System (SDS)
Version 2.3

Jeremy Bailey

Anglo-Australian Observatory

25 July 2001

Contents

1 Summary

The Self-Defining data system (SDS) is a system which allows the creation of self-defining hierarchical data structures in a form which allows the data to be moved between different machine architectures. Because, the structures are self-defining they can be used for communication between independent modules in a distributed system. The data structures are dynamic, allowing components to be added or deleted, or the size of arrays to be changed.

2 Introduction to SDS

SDS allows us to build essentially the same sort of data structures that we can build in most programming languages, for example in C using the `struct` keyword. A C struct groups together a number of items, each of which may be a simple variable, or may itself be another struct (thus giving rise to the hierarchical nature of the structures). Given this analogy, why do we need SDS? The reason is that SDS structures provide a number of features which are not present if we simply use the C struct. SDS structures have three important features:

- SDS structures are portable.
- SDS structures are dynamic.
- SDS structures are self-defining.

2.1 Portable Structures

Although it is easy to write a C struct to a file, and then read it back on another machine, the result will probably not be what was expected if the second machine is of a different architecture to the original. Differences may be encountered in the byte order of numeric items, in the representation of floating point numbers, and in the alignment requirements for structure components. The DEC VAX and SUN Sparc architectures, for example, differ in all these respects.

SDS is designed to look after all these architectural dependencies, and enable structures to be moved between machines, while guaranteeing to provide data in the correct format for the local machine.

2.2 Dynamic Structures

A C struct is a static structure, fixed at compile time. It is is not possible to dynamically add or delete components at run time. It is of course possible to create dynamic structures in C using pointers, linked lists etc., but such a structure is not then easily accessible as a single localized entity which can be written to a file or moved between machines.

SDS allows structures to be manipulated dynamically, while retaining the ability to move a structure as a single entity between machines.

2.3 Self-Defining Structures

If a data structure is to be passed between two communicating programs, then both programs need to have an identical copy of the structure definition to ensure they interpret the structure identically. In the case of a C struct this definition is the C source code declaring the structure. The same definition must be included in the source of both programs. For two tightly linked programs, it may not be too difficult

to ensure that the two programs are always using the same copy of the definition. For data moved around a widely distributed system, this can be much more difficult to accomplish, and it becomes very difficult to safely make any changes to a data structure without undesirable effects on other programs.

SDS solves this problem by including the definition of the structure with the data. This definition contains the name, type and dimensionality of each data item, and enables an item to be accessed by name, without the program necessarily having to know everything about the contents of the structure. It is thus much easier to develop communicating programs independently, since a new item may be added to a structure without requiring any changes in another program which reads that structure.

2.4 What SDS is

SDS consists of the SDS format defined in appendix F, and a library of C functions which are used for building and accessing SDS structures (appendix A). There is also a package of utility functions (Appendix B) and a Fortran interface to SDS (appendix D).

2.5 What SDS isn't

SDS is not a disk format, tape format, or any other device specific format. The SDS format simply specifies how a hierarchical structure can be encoded into an array of bytes. Given that in UNIX a file is simply a sequence of bytes, the representation of an SDS structure as a file is fairly obvious, but the SDS kernel itself includes no I/O operations.

SDS ascribes no meaning to the data contained in its structures. The use of the structure components is up to whatever higher level software calls SDS. The only thing SDS has to know about the data, is to recognize integer and floating point numbers which may need conversion for other architectures.

2.6 Comparison with XDR

There are other systems which address some of the problems which SDS solves. One of these is the eXternal Data Representation (XDR) standard used by the Sun RPC (Remote Procedure Call) system. XDR handles the problems of portability of data but falls short of the features provided by SDS in a number of ways.

- XDR is not a self-defining data format. Both sides of a connection must know the structure of the data in order to interpret it correctly, and serious problems could be encountered if the structure definitions at the two ends get out of step. This makes it more difficult to develop reliable modular software, particularly in cases where two communicating modules may be developed by independent programmers working at geographically separated locations.
- XDR handles portability by converting all data to a standard format. Thus, when moving data from a SUN to another SUN no data conversion is required since the standard format is already that for a Sun, but when moving data from a VAX to another VAX the data must first be converted to the standard format on one machine and then converted back to VAX format on the second machine. In the case of SDS no conversion is required in either of these cases, since data is simply flagged with its format, and converted only when it is read on a machine of different architecture.
- XDR is limited in the range of data types supported. In particular it does not support 16 bit integer types, nor does it support multi-dimensional arrays.

2.7 Comparison with HDS

Starlink's Hierarchical Data System (HDS), allows the building of hierarchical data structures similar to those of SDS, and in it's latest version supports portable structures across three different machine types, using the same approach as SDS, i.e. converting data only when it is read on a machine with different architecture to that on which it was written. HDS is self-defining, dynamic and portable. The differences between HDS and SDS are as follows:

- HDS constructs its data structures in disk files only. SDS by comparison manipulates data structures in memory. This is at least part of the reason that SDS is much faster than HDS (factors of 20 to 100 times faster for operations such as creating structures, navigating structures and reading and writing data). It also means that HDS structures cannot be transmitted between processes as messages or via shared memory as is possible with SDS.
- The definition of HDS is as a subroutine interface. The HDS format is nowhere defined. In contrast SDS has both a defined format and software for accessing the format.
- HDS has only a Fortran interface defined. SDS has both C and Fortran interfaces.
- HDS is written in a mixture of C and Fortran. This restricts its portability to machines which have a Fortran compiler and an implementation of the CNF package used to handle mixed language calls. SDS is written in pure C and can easily be ported to a wide range of systems, including systems which have no Fortran compiler, and which may not have access to disks, as will be the case for some real-time systems.

The SDS format is sufficiently compatible with the HDS format, that it is relatively straightforward to convert structures between the two formats, and programs to do this in both directions have been written (HDS2SDS and SDS2HDS).

2.8 Typical Use of SDS

Although SDS could be useful within the context of a single program, it really comes into its own when used to move data around a distributed system, particularly one which involves machines of different types.

Typically a program running on one machine would create a structure, write data into it and then write the structure to a file. A program running on another machine could then read the file, import the structure into SDS and read the data. If necessary it could further modify the structure before writing it out as a file again.

A file is only one possible way of moving an SDS structure between machines. Other possible ways would be by means of a message sent over the network, or by means of shared memory.

3 SDS Concepts

3.1 SDS Objects

An SDS structure is built out of three types of objects.

Primitive items - A primitive item is an item which can contain some data and may be either a scalar or an n dimensional array (where n $\leq$ 7) of one of the primitive types described below.

Structures - A structure is a list of named items each of which may be a primitive item, a structure or a structure array. The number of items may be between zero and 65535.

Structure Arrays - A structure array is an n-dimensional array (where n $\leq$ 7) of items, each of which is a structure.

We use the term *object* as a generic term for these three types of item. An object can range in complexity from a single primitive item, to a complex tree structure built up using structures and structure arrays.

3.2 Top Level Objects

A top level object is one that is not a member of any other structure or structure array. Most SDS functions can operate equally well on either top level objects, or on objects which are part of structures.

3.3 Primitive Types

There are nine types of primitive items as described in the following table.

SDS type code	C type	range
SDS_CHAR	char	
SDS_BYTE	signed char	-127 to 127
SDS_UBYTE	unsigned char	0 to 255
SDS_SHORT	short	-32767 to 32767
SDS_USHORT	unsigned short	0 to 65535
SDS_INT	int or long	-2147483647 to 2147483647
SDS_UINT	unsigned int or long	0 to 4294967295
SDS_I64	long (64 bit)	$-2^{63}-1$ to $2^{63}-1$
SDS_UI64	unsigned long (64 bit)	0 to $2^{64}-1$
SDS_FLOAT	float	machine dependent
SDS_DOUBLE	double	machine dependent

The types mostly correspond to the ANSI C types, and the ranges represent the minimum range guaranteed by ANSI C. It is possible that some SDS representations will support a greater range in some cases, but only numbers within the range listed above can be successfully transported between machines.

The SDS_INT and SDS_UINT types correspond to 32 bit integers. On many systems C int and long types are both 32 bits in length, but on some systems int may be a 16 bit type, and on some long may be a 64 bit type, so there is no C type which can be guaranteed to provide a 32 bit integer. The sds.h header files therefore define types INT32 and UINT32 which will provide a 32 bit integer type on any system, and will aid in writing portable SDS applications.

The SDS_I64 and SDS_UI64 types correspond to 64 bit integers. Only 64 bit architectures such as the Dec Alpha provide a 64 bit integer type in C. The sds.h header file defines types INT64 and UINT64 which can be used to hold a 64 bit integer. On 64 bit systems these types are equivalent to a long int. On 32 bit systems they will be defined as structs containing two 32 bit integers.

In C there is no distinction between a char type and a byte used to represent a number. SDS however, distinguishes between the two cases using the type SDS_CHAR to represent actual characters, and SDS_BYTE or SDS_UBYTE to represent byte length integers. This is required because future implementations may need to support character sets other than ASCII and will need to convert data between the different character sets, whereas byte length integers will not require such conversion.

3.4 SDS arrays

When SDS is used to create a primitive array, the array dimension information is stored in the structure, and a data block sufficient to hold the array will be used for the data. For example a 10 by 20 real array will cause a data block large enough to hold 200 real numbers to be used. SDS however does not ascribe any meaning to individual elements within the array, and has no functions which access specific elements by means of indices. The interpretation of the data is left to the high level software which makes used of SDS. SDS does not therefore enforce any particular index numbering scheme for arrays. The only functions which can access part of an array are `SdsPut` and `SdsGet` using the offset parameters, which specifies an offset into the data buffer treated as a one dimensional array, whatever the actual dimensionality.

Structure arrays have one (and only one) function which accesses array elements by indices and this is SdsCell. Array elements are numbered from 1 to n where n is the dimension specified on creation.

3.5 Internal and External Objects

SDS objects can exist in either internal or external forms. An internal structure has the following properties:

- An internal object exists in memory managed by SDS itself.
- The object is dynamic. All SDS functions are permitted including those which alter the structure, and delete or add components.
- The detailed representation of the structure is hidden from the user, and may be implementation dependent. The object can only be accessed through SDS functions.

External objects have the following properties:

- The objects exists in a data buffer supplied by the user.
- The object is static. Operations which read and write data, or navigate the structure are permitted, but operations which add or delete components or change the size of an item are not permitted.
- The structure is represented in the format described in appendix F of this document.

Objects are converted between external and internal form by the `SdsExport` and `SdsImport` functions.

3.6 Identifiers

SDS objects are referenced by the use of identifiers. An identifier is a variable which is passed to an SDS function to identify the object to operate on. Functions which create new SDS objects return identifiers to them. Identifiers can also be obtained using the functions which navigate structures. An identifier should be declared with the type **SdsIdType** defined in the include file **sds.h**.

3.7 Extra Information Field

Every SDS object contains in addition to the data, a field which can be used for any additional information which a higher level software package might want to associate with the object. The field is a character array of up to 128 bytes in length. It could be used, for example, to associate a type name with structured items (which all have the same type code of SDS_STRUCT), or to indicate the units of a numeric item.

This field is specified when the object is created, and can be subsequently modified and accessed using the `SdsPutExtra` and `SdsGetExtra` functions.

If the extra information field is not required its length should be specified as zero to minimize the space required by the object.

4 SDS Functions

4.1 Status Values

All SDS functions have a status argument which operates on the inherited status convention. The status argument should be set to the value SDS__OK (defined in sds.h) initially. If any SDS function is called with status set to any other value it will return immediately and do nothing. If an error is found during operation of an SDS function this is signalled by returning an appropriate error code in the status argument. The possible error codes returned by each function are listed in the function descriptions, and are defined in sds.h.

4.2 Structure Creation

New structures are created using the function `SdsNew`. To create a new top level object `SdsNew` is called with a `parent_id` of 0. The identifier returned can be used as the `parent_id` in subsequent calls to add components to a structure. `SdsNew` can be used to create structures, structure arrays and primitives.

When `SdsNew` is used to create a primitive object, the data for the object is initially in an undefined state, and no memory is allocated for the data. Memory only gets allocated when the data is accessed using `SdsPut` or `SdsPointer`. This *deferred creation* makes it possible to produce compact template objects which contain the definition of a structure, but no data.

```
/*  Structure creation example   */

#include "sds.h"

main(void)

{

  SdsIdType topid;   /* Top level identifier  */
  SdsIdType id1;     /* Identifier of first component  */
  SdsIdType id2;     /* Identifier of second component  */
  SdsIdType id3;     /* Identifier of third component  */
  unsigned long dims[2];   /*   Array dimensions   */
  long status;   /* Inherited status variable  */

/* Initialize status variable  */

  status = SDS__OK;

/* Create the top level object  */

  SdsNew(0, "Top", 0, NULL, SDS_STRUCT, 0, NULL, &topid, &status);
```

```
/* Create the first component - a primitive scalar integer */

  SdsNew(topid, "Comp1", 0, NULL, SDS_INT, 0, NULL, &id1, &status);

/* Create the second component - a two dimensional double array  */

  dims[0] = 10;
  dims[1] = 20;
  SdsNew(topid, "Comp2", 0, NULL, SDS_DOUBLE, 2, dims, &id2, &status);

/* Create the third component - a structure array - also illustrate the setting
   of the extra information field  */

  dims[0] = 4;
  SdsNew(topid, "Comp3", 17 ,"A Structure Array", SDS_STRUCT, 1, dims, &id3,
         &status);

/* Check everything is OK */

  if (status != SDS__OK) printf(" Error creating structure - %d\n",status);

  .
  .
```

4.3 Structure Navigation

These functions allow navigation through a structure tree and return identifiers to objects within it.

`SdsFind` is used to find a structure component by name and return an identifier to it.

`SdsIndex` is used to find a structure component by position. The position of a structure component is determined by the order in which the components were created using `SdsNew`. The first component has index number 1, the second 2, etc.

Generally `SdsFind` is the preferred way of finding structure components, since the self-defining nature of the structures is only fully used if components are accessed by name. However, `SdsIndex` is useful for programs which do not know what components to expect in a structure, for example, in a general program to list the contents of a structure.

The third navigation function is `SdsCell` which is used to find a component of a structure array and return an identifier to it.

4.4 Reading and Writing Data

Data is written into an SDS object using the function `SdsPut`. It copies data from a data buffer into the object. The object to be written into may be either a primitive, a structure or a structure array. If it is a structure the data buffer is assumed to contain a C struct equivalent to the SDS structure being written into. Equivalent means here having the same primitive components in the same order. `SdsPut` automatically skips over any padding regions which would be needed in the C struct to meet alignment requirements.

In the case of a primitive object, the offset parameter can be used to specify that the data will be written at some offset value into a primitive array. Thus `SdsPut` can be used to write only part of an array. The offset parameter is ignored when putting to a structure or structure array.

If the data for the primitive object is undefined, `SdsPut` causes the memory for the data to be allocated. Note, however, that it is not possible to do this in an external object, so data can only be written to an external object if the data is already defined.

Data is read back from an SDS object using `SdsGet`. This operates in a very similar way to `SdsPut` and can read data from primitives, structures or structure arrays.

```
/*  Example illustrating Structure get and put operations   */

#include <stdio.h>
#include "sds.h"

main(void)

{
long status;
SdsIdType topid,id1,id2,id3,id4;
unsigned long actlen;

/*  Define a C structure containing 4 items of different types  */
/*  The use of the INT32 type is necessary to ensure portability  */
/*  to all architectures  */

typedef struct block
   {
      char   c1;
      double d1;
      INT32  i1;
      float  f1;
   }  block;

block block1 = {'Q', 1.23456789, 9999, 3.1415926};
block block2;

/*  Create an SDS structure equivalent to the C structure   */

   status = SDS__OK;
   SdsNew(0,"test",0,NULL,SDS_STRUCT,0,NULL,&topid,&status);
   SdsNew(topid,"char1",0,NULL,SDS_CHAR,0,NULL,&id1,&status);
   SdsNew(topid,"double1",0,NULL,SDS_DOUBLE,0,NULL,&id2,&status);
   SdsNew(topid,"int1",0,NULL,SDS_INT,0,NULL,&id3,&status);
   SdsNew(topid,"float1",0,NULL,SDS_FLOAT,0,NULL,&id4,&status);

/*  Write the C structure (block1) into the SDS structure  */

   SdsPut(topid,sizeof(block),0,&block1,&status);

/*  Read from the SDS structure back into the C structure block2  */
```

```
   SdsGet(topid,sizeof(block),0,&block2,&actlen,&status);

/*  Print contents of block2  */

   printf(" %c %g %d %f \n",block2.c1,block2.d1,block2.i1,block2.f1);

}
```

An alternative way of accessing the data in an object is to use `SdsPointer`. This returns a pointer to the location of the data in the object itself. `SdsPointer` like `SdsPut` causes memory to be allocated for the data if it was previously undefined. SdsPointer can only be used with primitive items, not with structures or structure arrays.

When any change has been made to data which has been accessed via a pointer returned from `SdsPointer`, a call to `SdsFlush` must be made to ensure that the data is updated into the original structure. This is needed to handle cases where SDS has to provide a copy of the data rather than the original data in the structure.

```
void* ptr;

/*  Get a pointer to an item  */

   SdsPointer(id,&ptr,&status);
.
.  Make changes to item using pointer
.
/*  Flush the item to ensure it is updated in original structure  */

   SdsFlush(id,&status);
```

Note that a pointer can become invalid as a result of subsequent operations which may delete or move the data of an object such as `SdsDelete` or `SdsResize`. No checks are incorporated to guard against such problems.

4.5 Export and Import Functions

An internal SDS object is exported into a caller supplied buffer by the function `SdsExport`. Before exporting an object it is necessary to determine the size of buffer required to export it, and this can be obtained using `SdsSize`. This size can then be used to allocate a buffer of the required size (e.g. using `malloc`). Thus the following sequence could be used to export the object referenced by identifier id:

```
  unsigned long size;
  void *buffer;
  long status;
  SdsIdType id;

  status = SDS__OK;
  SdsSize(id,&size,&status);
  buffer = malloc(size);
```

```
if (buffer != NULL)
  SdsExport(id,size,buffer,&status);
```

An exported object can be reimported into SDS using `SdsImport`. This function returns an identifier to a new internal copy of the object which can subsequently be accessed or manipulated using any SDS function.

In many cases, however, it will not be necessary to import an object in order to access it. The function `SdsAccess` returns an identifier to an object contained in a buffer, but accesses it in place as an external object. Operations which read and write data and navigate the structure are permitted on such an external object, but operations which alter the structure are not permitted.

If an SDS item contains undefined primitive data (i.e. items that have been created with `SdsNew`, but not had any data yet written to them, then these items are undefined and occupy no space in the exported item, they cannot be written to until the structure is reimported. This feature allows compact template objects to be created for structures which contain large data arrays. If this behaviour is not wanted, there is an alternate export function `SdsExportDefined` which exports an object, allocating space for all primitve objects even those which were originally undefined. Use `SdsSizeDefined` to get the buffer size needed for this function.

4.6 Editing Structures

The function `SdsDelete` deletes an object. If the object is a component of a structure, then subsequent components in the structure are shifted down by one in position.

Deletion, like all the operations in this section, is to be understood as a recursive operation deleting all the components of the object, if it is a structure.

When an object has been deleted any identifiers which refer to it or its components become invalid, and any attempt to use them will result in an SDS__BADID status being returned.

The function `SdsCopy` makes a complete copy of an object as a new top level object. The original object may be either external or internal. The copy is always internal.

`SdsExtract` extracts an object from a structure. The extracted object becomes an independent top level object, and is deleted from the original structure.

`SdsInsert` inserts an object into a structure. The object to be inserted must be a top level object (it is not permitted in SDS for the same object to be a member of two different structures). The object is inserted at a position following all the existing objects in the structure.

`SdsResize` changes the number and/or size of the dimensions of an array. This operation can be performed on primitive arrays or structure arrays. Note that `SdsResize` does not move the data elements in the storage representing the array, so there is no guarantee that after resizing the array the same data will be found at the same array index positions as before resizing, though this will be the case for simple changes such as a change in the last dimension only.

`SdsRename` is used to change the name of an object.

4.7 Freeing Identifiers

Each identifier used by SDS requires allocation of resources within SDS. Allocation of a very large number of identifiers can cause degradation of performance. The function `SdsFreeId` can be used to free up resources allocated to an identifier so that they can be reallocated to a subsequent identifier.

4.8 Other Functions

`SdsInfo` returns the name and type code of an object. If it is an array it also returns the array dimensions.

`SdsIsExternal` is used to enquire whether an SDS object os external. `SdsExternInfo` returns the address of the buffer containing an external item (i.e. the address which was originally given to `SdsAccess`

`SdsGetExtra` and `SdsPutExtra` are used to read and write the extra information field of the object. Note that if the object is external, `SdsPutExtra` may not increase the number of bytes in the field.

5 The SDS Utility Package

The functions described in the previous section form the SDS kernel. The kernel is designed to be implemented with minimal system requirements, and in particular includes no I/O functions. The SDS utility package includes some additional functions which are implemented in terms of the kernel functions.

`SdsList` is used to list the contents of an SDS structure on the standard output stream. The listing includes the name and type of each item, and the value of primitive items (the first few values only for arrays).

`SdsWrite` is used to write an SDS structure to a file. The file can then be read back using `SdsRead`.

The following example program uses SdsRead and SdsList to list the contents of an SDS file.

```
#include "sds.h"

main(int argc, char* argv[])

{
  long status;
  long id;

  if (argc > 1)
    {
      status = SDS__OK;
      SdsRead(argv[1],&id,&status);
      SdsList(id,&status);
    }
}
```

6 The SDS Compiler

A C structure can be put and retrieved from a similar SDS structure using SdsPut and SdsGet. This makes it desirable to be able to automatically generate SDS calls to produce an SDS structure equivalent to the C structure. `sdsc` does this job.

`sdsc` first runs its input through a C preprocessor. During this the macros '`SDS`' and '`___SDS___`' will be defined (in addition to any macros defined by default). The result should be a series of C definitions followed by one and only one structure declaration.

For example:

```
typedef long int mytype ;

typedef struct honey {
                long int iiii; } pppp;

struct mystruct { long int jenny ;
                char john ;
                unsigned char fred ;
                pppp p;
                mytype aaaa ;
                } jim;
```

In this example, the declaration of the structure `jim`, of type "mystruct" is the required structure. The `typedef`'s define some of its elements. (Note, the required definitions could be extracted from C source code by use of the `SDS` or `___SDS___` macros).

The result of running this though sdsc is a C routine body. In this case we get (minus some comments)-

```
{
    int tid0 = 0;
    int id = 0;
    if (*status != SDS__OK) return;

    SdsNew(0,"mystruct", 0 , NULL, SDS_STRUCT, 0, NULL, &tid0, status);
    SdsNew(tid0, "jenny", 0, NULL, SDS_UINT, 0 , NULL,  &id, status);
    SdsNew(tid0, "john", 0, NULL, SDS_CHAR, 0 , NULL,  &id, status);
    SdsNew(tid0, "fred", 0, NULL, SDS_BYTE, 0 , NULL,  &id, status);
    {
        int tid1 = 0;
        SdsNew(tid0,"p", 0 , NULL, SDS_STRUCT, 0, NULL, &tid1, status);
        SdsNew(tid1, "iiii", 0, NULL, SDS_UINT, 0 , NULL,  &id, status);
    }
    SdsNew(tid0, "aaaa", 0, NULL, SDS_UINT, 0 , NULL,  &id, status);

    return(tid0);
}
```

The user should provide the routine declaration. A full example of users' code follows-

```
/* An example illustrating the used of sdsc */
struct mystruct { long int fred;
                  unsigned char john;
                } aaaa;


/* This rest is not passed to sdsc, only to the C compiler */
#ifndef SDS
#include "sds.h"
#include "stdio.h"

/* CreateStruct uses the result of running this module through sdsc */
/* These results should be put in the file test2.h      */
int CreateStruct(int *status)
```

```
#include "test2.h"

/* Now the main routine  */
struct mystruct jjj = { 1, 'c' };
int main()

{
    int status = SDS__OK;
    int id;
    id = CreateStruct(&status);

    SdsPut(id,sizeof(struct mystruct), 0, &jjj, &status);
    SdsList(id,&status);
    return(0);
}

#endif
```

To make the above you would use the commands-

```
sdsc test2.c test2.h
gcc -o test2 test2.c
```

There is also a callable interface to the compiler. This takes as its input a string containing the structure definition.

```
/*  Example illustrating the use of SdsCompiler() */

#include "sds.h"
#include "stdio.h"
int main()
{
    struct thestruct {
                long int john;
                char fred;
  };

    struct thestruct b1 = { 10, 'c' };
    struct thestruct b2 = { 0,0};
    int status  = 0;
    int id;
    SdsCompiler("struct thestruct { long int john ; char fred ; } i;",
                 1, &id, &status);

    if (status != 0)
        fprintf(stderr, "sdscompiler returned a status of %d\n",status);
    else
    {
        int actlen;
        printf("Contents should be - %d, %c\n",b1.john,b1.fred);
        SdsPut(id,sizeof(struct thestruct),0,&b1,&status);
        SdsList(id,&status);
        SdsGet(id,sizeof(struct thestruct),0,&b2,&actlen,&status);
```

```
        printf("Contents are - %d, %c\n",b2.john,b2.fred);
        if (status != 0)
            fprintf(stderr, "Put/Get returned a status of %d\n",status);
    }
    return(status);
}
```

7 The Arg Functions - A simple interface to SDS

7.1 Introduction

The SDS functions to create and access structures have quite a large number of parameters because of the large number of features they provide. In many cases the complexity of using SDS calls directly will not be necessary because a higher level package, layered on top of SDS is also available. This is the Arg package, so called because it is used to access the arguments of actions in the AAO 2dF software system. It is however nothing more than a simple interface to SDS.

ARG differs from SDS in the following ways:

- In SDS items are referenced by means of identifiers. In ARG items are referenced by means of the parent identifier and item name. This is slightly less efficient, but means that it is not necessary to first get the identifier by means of a `SdsFind` call.
- In SDS an item must be seperately created and then written to. In ARG an item will be created automatically, if necessary, in the same operation as writing a value.
- ARG performs type conversion if necessary between the actual type of an item and the type of the value being written or read.
- ARG currently only supports scalar items and character strings. More complex objects must be accessed by direct SDS calls.

ARG may be used on its own, or SDS calls may be interspersed with ARG calls.

7.2 Arg Functions

The Arg library provides three types of functions:

ArgNew - This is used to create a new argument structure.

ArgPutx - These functions write items into a argument structure. If the item does not yet exist it is created.

ArgGetx - These functions read items from an argument structure.

The `ArgPutx` and `ArgGetx` functions have versions to read and write scalar items of each of the possible ANSI C types, and there are additional functions to read and write character strings.

The functions are as follows:

Type	Put Function	Get Function
char	ArgPutc	ArgGetc
short	ArgPuts	ArgGets
unsigned short	ArgPutus	ArgGetus
long	ArgPuti	ArgGeti
unsigned long	ArgPutu	ArgGetu
float	ArgPutf	ArgGetf
double	ArgPutd	ArgGetd
char[]	ArgPutString	ArgGetString

The ArgPutx and ArgGetx function perform type conversion if required between different arithmetic types (i.e. if the item in the structure has a different type to that specified by the calling function). Conversion follows the ANSI C rules for arithmetic type conversion. If the item is out of range of the destination type the ARG__CNVERR status will be returned, and the operation will not be completed. The Arg functions will aslo perform conversion between character strings and numeric types. Full descriptions of these functions are given in appendix C.

7.3 Arg Example

```
#include "sds.h"
#include "arg.h"

main(void)

{

  SdsIdType id;    /* Sds identifier  */
  long status;     /* Inherited Status Variable  */

  status = SDS__OK;

/*  Create a structure  */

  ArgNew(&id,&status);

/*  Create an integer component  */

  ArgPuti(id, "Comp1", 123, &status);

/*  Create a float component  */

  ArgPutf(id, "Comp2", 3.141592, &status);

/*  Create a string component  */

  ArgPutString(id, "Comp3", "This is a string", &status);

}
```

8 The Fortran Interface

A Fortran Interface to SDS has been supplied in addition to the standard C interface already described. The Fortran interface differs from the C version in the following ways:

- The Fortran subroutines have names similar to those of the corresponding C functions but are all in upper case and use an underscore to separate words (e.g. SDS_PUT_EXTRA for SdsPutExtra).
- The Fortran versions use Starlink's error message system (EMS) to report errors. The C version does not use EMS which would restrict its portability.
- The functions SdsGet and SdsPut can not easily be converted to standard Fortran versions, since they handle data items of various different types. Therefore a set of additional routines have been provided to get and put the standard Fortran data types (SDS_GETC, SDS_GETD, SDS_GETI etc.). In addition SDS_GET and SDS_PUT which use the non-standard Fortran BYTE type and can be used to return values of any the SDS types.

The Fortran interface has been created using Starlink's CNF package to handle portable mixed language programming. This has so far been implemented on VAX, SUN and DECstation, and is thus not as portable as the SDS kernel and utilities which are in pure C.

The Fortran interface also includes the SDS utility functions and the ARG functions. The Fortran equivalents of the ARG functions are a little different to the C versions as they are based on Fortran rather than C types.

The SDS Fortran subroutines are described in appendix D.

9 The Implementation

The current implementation of SDS is written in portable C and should compile on any machine with an ANSI compliant C compiler, and will in fact compile with many compilers which are not fully ANSI compliant. It does require a compiler which supports function prototypes.

The implementation makes the following assumptions about the storage architecture. Chars must be 8 bits in length, short integers must be 16 bits in length, long integers and floats must be 32 bits in length, doubles must be 64 bits in length. No assumptions are made about the length of ints and pointers. An architecture that did not conform to these assumptions would require a specialized implementation.

Conditional compilations are used to set the value of the array `local_format` which contains the format codes for each data type to an appropriate value for the machine. There are basically three options used at present:

- Little endian integers and VAX floating point format (DEC VAX).
- Little endian integers and byte swapped IEEE floating point format (DECstation and Intel 80x86).
- Big endian integers and IEEE floating point format (SUN Sparcstation and Motorola 680x0)

The alignment requirements for different machines are not specified in conditional compilations, but are determined by tests built into the code. The alignment requirements for each of the types, short, long, float, double are determined by these tests. Three cases have been encountered in practice.

- No alignment. Any item can begin at any byte address (DEC VAX).

- Even alignment. All items larger than char are aligned to even addresses. (Intel 80x86 and Motorola 680x0).
- Full Alignment. Items are aligned to their full size, i.e. doubles to 8 byte multiples, longs and floats to 4 bytes, shorts to 2 bytes. (Sun Sparcstation and DECstation).

SDS has been compiled and tested on the following machines:

- DEC VAX running VMS - Using the VAX C compiler.
- DEC Alpha running OSF/1 - Using the Dec C compiler or the GNU C compiler.
- DEC Alpha runing VMS.
- SUN Sparcstation running SunOs 4 - Using the GNU C compiler.
- SUN Sparcstation running Solaris 2 - Using the Ansi C SparcCompiler.
- DECstation - Using the DECstation C compiler.
- Apple Macintosh (68020/30) - Using the MPW C compiler, Symantec Think C compiler, or Symantec C++ compiler or Metroworks CodeWarrior.
- 68020/30 VME system running VxWorks - Using a GNU C cross compiler running on a Sparcstation.
- IBM PC - Using the Microsoft C compiler.

SDS structure have been successfully moved between all these machines.

10 SDS version 2.2 Release

The SDS version 2.2 release includes the following:

- The SDS kernel (functions described in appendix A)
- The SDS utility functions (appendix B)
- The ARG functions (appendix C)
- The SDS compiler.
- The `sdstest` test program. This performs an exhaustive test of all SDS functions, and tests operations on scalars and arrays of all data types in both internal and external cases.
- The `readtest` test program. This tests the ability of SDS to read 'foreign' data files (i.e. SDS data files created on machines of different architecture).
- Three SDS data files created on VAX, SUN Sparcstation and DECstation for use with `readtest` (These test both big and little endian cases, and IEEE and VAX floating point formats).
- The `sdslist` program which is used to list the contents of an SDS file.
- The `sdstimes` program which performs some timing tests on SDS functions.

The VAX/VMS, SUN and DECstation releases includes in addition the SDS Fortran interface (Appendix D) and the HDS2SDS and SDS2HDS conversion programs. These should be made available for the DECstation in a future release.

10.1 Special Considerations when using SDS under VxWorks

The VxWorks real time operating system differs from operating systems such as Unix in providing a single address space shared by all processes. In such a system there is thus a single SDS context shared by all processes, rather than each process having its own context as on Unix. The SDS code has to be reentrant, and this impacts on the operation of the facility for allocating identifiers used by SDS. In VxWorks this becomes a shared resource and has to be protected by means of a VxWorks semaphore. There is thus a possibility that an SDS function which allocates a new identifier will have to wait to gain access. It is advisable not to call such functions from interrupt level code.

In VxWorks it is also possible for an SDS object to be accessed by more than one task. Although this will work, SDS includes no checks to prevent objects being corrupted by simultaneous access from more than one task. Programs using SDS in this way should arrange their own access control mechanism for the shared objects (e.g. using semaphores).

11 History

11.1 Changes in Version 2.2

The following new functions were added; `SdsExportDefined`, `SdsSizeDefined`, `SdsIsExternal`, `SdsExternInfo`.

11.2 Changes in Version 2.1

Support for safe operation in a POSIX multithreaded environment. POSIX mutex semaphores are used to control access to the allocation of identifiers. To get this behaviour SDS must be compiled with the macro POSIX_THREADS defined.

Fixed a bug in the semaphore protection of id allocation under VxWorks.

11.3 Changes in Version 2.0

Support for G floating point type (needed to support VMS on Alpha).

Several bug fixes to eliminate memory leaks.

11.4 Changes in Version 1.4

- This version supports Dec Alpha systems running OSF/1. A number of changes have been made to aid portability to 64 bit architectures.
- 64 bit integer and unsigned integer types have been added.
- A bug has been fixed in SdsRead which resulted in the file not being closed.

11.5 Changes in Version 1.3

- The arg package now supports conversion between character strings and numeric types.
- The SdsSize and SdsExport operations are now permitted on external objects.
- A bug has been fixed in SdsGet which could cause incorrect data to be returned when 2, 4 or 8 bytes were being transferred and the destination variable was not aligned on a corresponding boundary.
- A bug has been fixed in HDS2SDS which caused character string items to be transferred incorrectly on the VAX.

11.6 Changes in Version 1.2

- The SDS compiler (written by Tony Farrell) has been included. It is available both as a standalone program and a callable function.
- The Fortran interface and SDS to HDS conversion program have been added to the DECstation release of SDS.
- The ARG functions (and Fortran interface to ARG) have been added to the SDS release.
- Some function prototypes were missing from the include file in the previous version. These have now been added.

11.7 Changes in Version 1.1

- The type names for identifiers and type codes are now, `SdsIdType` and `SdsCodeType`.
- The VxWorks release of SDS is now available.
- The SUN 4 release now includes the Fortran interface and SDS to HDS conversion programs.
- Make files have been modified so that `make` alone will do a complete rebuild.

A SDS Kernel Function Descriptions

A.1 SdsAccess — Return an identifier to an external object

Function: Return an identifier to an external object

Description: Make an external object (one exported by SdsExport) accessible to SDS by returning an identifier to it.

Any SDS operations which do not change the structure of the object may be performed on the external object. These include navigation operations (SdsFind, SdsIndex, SdsCell), data access operations (SdsGet, SdsPut, SdsPointer) and inquiry operations (SdsInfo).

Operations which are not permitted on an external object are those which add or remove components (SdsNew, SdsDelete), or write operations (SdsPut or SdsPointer) to data items which are currently undefined.

Unlike SdsImport, SdsAccess does not make a copy of the object. The object is accessed in place in the original buffer.

Language: C

Declaration: void SdsAccess(void *data, SdsIdType *id, StatusType * SDSCONST status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	data	void*	The buffer containing the object to be accessed.
<	id	long*	Identifier of the external object.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__NOTSDS	Not a valid SDS object.
SDS__NOMEM	Insufficient memory.

Support: Jeremy Bailey, AAO

Version date: 18-Oct-91

A.2 SdsCell — Find component of a structure array

Function: Find component of a structure array

Description: Given the indices to a component of a structure array, return an identifier to the component.

Language: C

Declaration: void SdsCell(SdsIdType array_id, long nindices, unsigned long *indices, SdsIdType *id, StatusType * SDSCONST status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	array_id	SdsIdType	Identifier of the structure array.
>	nindices	long	Number of indices supplied in the array indices. This should be one or the same as the number of dimensions of the array.
>	indices	unsigned long*	An array of length nindices containing the indices to the component of the structure array. If nindicies is 1, then treat the structure array as having only one dimension even if it has more.
<	id	SdsIdType*	The identifier of the component.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__NOMEM	Insufficient memory for creation
SDS__NOTARRAY	Not a structure array.
SDS__INDEXERR	Indices invalid.

Support: Jeremy Bailey, AAO

Version date: 18-Oct-91

A.3 SdsCopy — Make a copy of an object

Function: Make a copy of an object

Description: Make a copy of an object and return an identifier to the copy. The copy is a new top level object, the original object is unchanged by the operation.

The object being copied can be either external or internal. The copy is always an internal object.

Language: C

Declaration: void SdsCopy(SdsIdType id, SdsIdType *copy_id, StatusType * SDSCONST status)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

>	id	SdsIdType	Identifier of the object to be copied.
<	copy_id	SdsIdType*	Identifier of the copy.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__BADID	The identifier is invalid.
SDS__NOMEM	Insuficient memory.

Support: Jeremy Bailey, AAO

Version date: 23-Oct-91

A.4 SdsDelete — Delete an object

Function: Delete an object

Description: Delete an object, freeing any memory associated with it. Subsequent attempts to access the object through any identifier associated with it will return the SDS__BADID status. A structure array element cannot be deleted. An attempt to do so will result in the SDS__ILLDEL status.

Deleting an object does not free the memory associated with the identifier referencing it. This memory can be freed with the SdsFreeId function.

Language: C

Declaration: void SdsDelete(SdsIdType id, long *status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	id	SdsIdType	Identifier of the object to be deleted.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__BADID	The identifier is invalid.
SDS__EXTERN	Object is external.
SDS__ILLDEL	Object cannot be deleted.

Support: Jeremy Bailey, AAO

Version date: 18-Oct-91

A.5 SdsExport — Export an object into an external buffer

Function: Export an object into an external buffer

Description: Export an object into an external buffer.

Once exported an object can be moved around in memory, written to a file etc., and subsequently returned to the SDS system either by using SdsImport to import it back into the system, or SdsAccess, to access it as an external object.

The original internal version of the object continues to exist, in addition to the external copy. All identifiers to the object continue to refer to the original internal copy.

With SdsExport, any undefined primitive data items occupy no space in the exported item, and cannot be written or read until the item is reimported. This enables the creation of compact templates for structures which may contain large arrays. If this behaviour is not wanted use SdsExportDefined, which allocates full space in the external structure for undefined primitive items.

The length of the buffer required for SdsExport can be determined by a call to SdsSize.

Language: C

Declaration: void SdsExport(SdsIdType id, unsigned long length, void *data, StatusType * SDSCONST status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	id	SdsIdType	Identifier of the structure to be exported.
>	length	unsigned long	Size in bytes of the buffer.
<	data	void*	The buffer into which the object will be exported.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__BADID	The identifier is invalid.
SDS__TOOLONG	The object is too large for the buffer
SDS__EXTERN	The object is external.

Support: Jeremy Bailey, AAO

Version date: 18-Oct-91

A.6 SdsExportDefined — Export an object into an external buffer

Function: Export an object into an external buffer

Description: Export an object into an external buffer.

Once exported an object can be moved around in memory, written to a file etc., and subsequently returned to the SDS system either by using SdsImport to import it back into the system, or SdsAccess, to access it as an external object.

The original internal version of the object continues to exist, in addition to the external copy. All identifiers to the object continue to refer to the original internal copy.

SdsExportDefined allocates space in the external item for undefined data items, so that these can have their values filled in later by an SdsPut (or SdsPointer) to the external item.

The length of the buffer required for SdsExportDefined can be determined by a call to SdsSizeDefined.

Language: C

Declaration: void SdsExportDefined(SdsIdType id, unsigned long length, void *data, StatusType * SDSCONST status)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

>	id	SdsIdType	Identifier of the structure to be exported.
>	length	unsigned long	Size in bytes of the buffer.
<	data	void*	The buffer into which the object will be exported.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__BADID	The identifier is invalid.
SDS__TOOLONG	The object is too large for the buffer
SDS__EXTERN	The object is external.

Support: Jeremy Bailey, AAO

Version date: 13-Jul-98

A.7 SdsExternInfo — Return the address of an external object

Function: Return the address of an external object

Description: Return the address of an external SDS object given its id. This is the address which was given to SdsAccess.

Language: C

Declaration: void SdsExternInfo(SdsIdType id, void ** address, StatusType * SDSCONST status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	id	SdsIdType	Identifier of the object
<	address	void **	Address of object
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__BADID	The identifier is invalid.
SDS__EXTERN	Not an external object.

Support: Jeremy Bailey, AAO

Version date: 13-Jul-98

A.8 SdsExtract — Extract an object from a structure

Function: Extract an object from a structure

Description: Extract an object from a structure. The extracted object becomes a new independent top level object. The object is deleted from the original structure.

The identifier must not be that of a structure array component.

If the identifier is already that of a top level object, then the function does nothing.

Language: C

Declaration: void SdsExtract(SdsIdType id, StatusType * SDSCONST status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	id	SdsIdType	Identifier of the object to be extracted.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__BADID	The identifier is invalid.
SDS__ILLDEL	Object cannot be extracted.
SDS__EXTERN	The object is external.

Support: Jeremy Bailey, AAO

Version date: 23-Oct-91

A.9 SdsFind — Find a structure component by name

Function: Find a structure component by name

Description: Given the name of a component in a structure, return an identifier to the component.

Language: C

Declaration: void SdsFind(SdsIdType parent_id, char *name, SdsIdType *id, StatusType * SDSCONST status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	parent_id	SdsIdType	Identifier of the structure.
>	name	char*	Name of the component to be found.
<	id	SdsIdType*	Identifier to the component.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__NOMEM	Insufficient memory for creation
SDS__NOTSTRUCT	parent_id not a structure
SDS__NOITEM	No item with that name

Support: Jeremy Bailey, AAO

Version date: 18-Oct-91

A.10 SdsFlush — Flush data updated via a pointer

Function: Flush data updated via a pointer

Description: If a primitive data item is accessed via SdsPointer, and the data array updated via the returned pointer, then SdsFlush must be called to ensure that the data is updated in the original structure.

This must be done since implementations on some machine architectures may have to use a copy of the data rather than the actual data when returning a pointer.

Language: C

Declaration: void SdsFlush(SdsIdType id, StatusType * SDSCONST status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	id	SdsIdType	Identifier of the primitive item.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__BADID	Identifier invalid
SDS__NOTPRIM	Not a primitive item

Support: Jeremy Bailey, AAO

Version date: 7-Feb-92

A.11 SdsFreeId — Free an identifier, so that its associated memory may be reused.

Function: Free an identifier, so that its associated memory may be reused.

Description: Each identifier allocated by SDS uses memory. To avoid excessive allocation of memory the SdsFreeId function can be used to free the memory associated with an identifer that is no longer needed. When this is done the memory will be reused by SDS for a subsequent identifier when needed.

Language: C

Declaration: void SdsFreeId(SdsIdType id, StatusType * SDSCONST status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	id	SdsIdType	Identifier to be freed
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__BADID	The identifier is invalid.

Support: Jeremy Bailey, AAO

Version date: 23-Jan-92

A.12 SdsGet — Read the data from an object

Function: Read the data from an object

Description: The object may be a primitive item or a structure or structure array. Read the data from an item into a buffer. If the object is primitive data is transferred starting at the position in the item specified by offset, until the buffer if filled, or the end of the data array is reached.

If the object is a structure or structure array, the data from all its primitive components are copied into the buffer in order of their position in the structure. Alignment adjustments are made as necessary to match the alignment of an C struct equivalent to the SDS structure. (Since these alignment requirements are machine dependent the actual sequence of bytes returned could be different on different machines). In the structure or structure array case the offset parameter is ignored.

If necessary (e.g. if the data originated on a machine with different architecture) the data for the object is converted (in place) from the format stored in the data item to that required for the local machine.

Language: C

Declaration: void SdsGet(SdsIdType id, unsigned long length, unsigned long offset, void *data, unsigned long *actlen, StatusType * SDSCONST status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	id	SdsIdType	Identifier of the object.
>	length	unsigned long	Length in bytes of the buffer supplied to receive the data.
>	offset	unsigned long	Offset into the data object at which to start reading data. The offset is measured in units of the size of each individual item in the array - e.g. 4 bytes for an `INT` or 8 bytes for a `DOUBLE`. The offset is zero to start at the beginning of the array. This parameter is ignored if the object is a structure or structure array.
<	data	void*	Buffer to receive the data.
<	actlen	unsigned long*	Actual number of bytes transferred.
!	status	StatusType*	Modified status. Possible failure codes are:

`SDS__BADID`	Invalid identifier
`SDS__UNDEFINED`	Data undefined

Support: Jeremy Bailey, AAO

Version date: 18-Oct-91

A.13 SdsGetExtra — Read from the extra information field of an object.

Function: Read from the extra information field of an object.

Description: Read bytes from the extra information field of an object. Bytes are copied until the supplied buffer is filled up or until all bytes in the field are copied.

Language: C

Declaration: void SdsGetExtra(SdsIdType id, long length, char* extra, unsigned long* actlen, StatusType `*` `SDSCONST` status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

>	id	SdsIdType	Identifier of the object
>	length	long	Length of buffer to receive data.
<	extra	char*	Buffer to receive the extra information copied from the object.
<	actlen	unsigned long*	actual number of bytes copied.
!	status	StatusType*	Modified status. Possible failure codes are:

`SDS__BADID`	The identifier is invalid.

Support: Jeremy Bailey, AAO

Version date: 24-Oct-91

A.14 SdsImport — Import an object from an external buffer

Function: Import an object from an external buffer

Description: Import an object from an external buffer and return an identifier to the internal copy created. The object must have been previously exported using SdsExport.

The original external version of the structure continues to exist, in addition to the internal copy.

A fully dynamic internal structure is created in which all SDS operations are valid. However, to merely access the data in an object SdsAccess can be used in place of SdsImport.

Language: C

Declaration: void SdsImport(void *data, SdsIdType *id, StatusType * SDSCONST status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	data	void*	The buffer from which the object will be imported.
<	id	SdsIdType*	Identifier of the new internal object.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__NOTSDS	Not a valid sds object.
SDS__NOMEM	Insufficient memory.

Support: Jeremy Bailey, AAO

Version date: 18-Oct-91

A.15 SdsIndex — Find a structure component by position

Function: Find a structure component by position

Description: Given the index number of a component in a structure, return an identifier to the component.

Language: C

Declaration: void SdsIndex(SdsIdType parent_id, long index, SdsIdType *id, StatusType * SDSCONST status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	parent_id	SdsIdType	Identifier of the structure.
>	index	long	Index number of the component to be returned. Items in a structure are numbered in order of creation starting with one.
<	id	SdsIdType*	Identifier to the component.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__NOMEM	Insufficient memory for creation
SDS__NOTSTRUCT	parent_id not a structure
SDS__NOITEM	No item with that index number

Support: Jeremy Bailey, AAO

Version date: 18-Oct-91

A.16 SdsInfo — Return information about an object

Function: Return information about an object

Description: Given the identifier to an object, return the name, type code and dimensions of the object.

Language: C

Declaration: void SdsInfo(SdsIdType id, char *name, SdsCodeType *code, long *ndims, unsigned long *dims, StatusType * SDSCONST status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	id	SdsIdType	The identifier to the data object.
<	name	char*	The name of the data object. A pointer to a character string with space for at least 16 characters should be used.
<	code	SdsCodeType*	The type code for the object. One of the following values (defined in sds.h):

SDS_STRUCT	Structure
SDS_CHAR	Character
SDS_BYTE	Signed byte
SDS_UBYTE	Unsigned byte
SDS_SHORT	Signed short integer
SDS_USHORT	Unsigned short integer
SDS_INT	Signed long integer
SDS_UINT	Unsigned long integer
SDS_I64	Signed 64 bit integer
SDS_UI64	Unsigned 64 bit integer
SDS_FLOAT	Floating point
SDS_DOUBLE	Double precision floating point

<	ndims	long*	The number of dimensions if the object is a primitive or structure array.
<	dims	unsigned long*	The dimensions of the data. An array of size at least 7 should be allowed to receive this.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__BADID	The identifier is invalid

Prior requirements: None.

Support: Jeremy Bailey, AAO

Version date: 18-Oct-91

A.17 SdsInsert — Insert an existing object into a structure

Function: Insert an existing object into a structure

Description: An existing top level object is inserted into a structure.

Language: C

Declaration: void SdsInsert(SdsIdType parent_id, SdsIdType id, StatusType * SDSCONST status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	parent_id	SdsIdType	The identifier of the structure to which the component is to be added.
>	id	SdsIdType	The identifier of the object to be inserted.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__BADID	Invalid identifier
SDS__NOTSTRUCT	Parent is not a structure
SDS__NOTTOP	Not a top level object
SDS__NOMEM	Insufficient memory
SDS__EXTERN	Object is external

Support: Jeremy Bailey, AAO

Version date: 23-Oct-91

A.18 SdsInsertCell — Insert object into a structure array

Function: Insert object into a structure array

Description: Insert a top level object into a structure array at a position specified by its indices. Delete any object that is currently at that position.

Language: C

Declaration: void SdsInsertCell(SdsIdType array_id, long nindices, unsigned long *indices, SdsIdType id, StatusType * SDSCONST status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	array_id	SdsIdType	Identifier of the structure array.
>	nindices	long	Number of indices supplied in the array indices. This should be 1 or the same as the number of dimensions of the array.
>	indices	unsigned long*	An array of length nindices containing the indices to the component of the structure array. If nindicies is 1, then treat the structure array as having only one dimension even if it has more.
>	id	SdsIdType	The identifier of the component to be inserted.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__EXTERN	Structure array or object is external.
SDS__NOTARRAY	Not a structure array.
SDS__INDEXERR	Indices invalid.
SDS__NOTTOP	Not a top level object.

Support: Jeremy Bailey, AAO

Version date: 24-Aug-96

A.19 SdsIsExternal — Enquire whether an object is external

Function: Enquire whether an object is external

Description: Enquire whether an object is external

Language: C

Declaration: void SdsIsExternal(SdsIdType id, int *external, StatusType * SDSCONST status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	id	SdsIdType	Identifier of the object
<	external	int *	Non zero if the object is external
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__BADID	The identifier is invalid.

Support: Jeremy Bailey, AAO

Version date: 13-Jul-98

A.20 SdsNew — Create a new object

Function: Create a new object

Description: Creates a new component in an existing internal structure or a new top level object. A top level object is created by specifying a parent_id of zero. The new object can be a structure, a structure array, or a primitive. A structure array is specified by means of a type code of SDS_STRUCT and a non-zero number for ndims. If the type code is SDS_STRUCT and ndims is zero an ordinary structure is created. A primitive type is specified by the appropriate type code.

Language: C

Declaration: void SdsNew(SdsIdType parent_id, char *name, long nextra, char *extra, SdsCodeType code, long ndims, unsigned long *dims, SdsIdType *id, StatusType * SDSCONST status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	parent_id	SdsIdType	The identifier of the structure to which the object is to be added. Use a value of zero to create a new top level object.
>	name	char*	The name of the object to create. The name should be of maximum length 16 characters including the terminating null.
>	nextra	long	The number of bytes of extra information to be included (maximum 128).
>	extra	char*	The extra information to be included with the item. nextra bytes from here are copied into the structure.
>	code	SdsCodeType	The type code for the item to be created. One of the following values (defined in sds.h):

SDS_STRUCT	Structure
SDS_CHAR	Character
SDS_BYTE	Signed byte
SDS_UBYTE	Unsigned byte
SDS_SHORT	Signed short integer
SDS_USHORT	Unsigned short integer
SDS_INT	Signed long integer
SDS_UINT	Unsigned long integer
SDS_I64	Signed 64 bit integer
SDS_UI64	Unsigned 64 bit integer
SDS_FLOAT	Floating point
SDS_DOUBLE	Double precision floating point

>	ndims	long	Number of dimensions for the item. Zero to create a scalar item.
>	dims	unsigned long*	Array of dimensions for the item. Should be of size at least ndims. A NULL pointer may be used if the item is a scalar.
<	id	SdsIdType*	Identifier to the created object.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__BADID	Invalid identifier
SDS__NOMEM	Insufficient memory for creation
SDS__LONGNAME	name is too long
SDS__EXTRA	Too much extra data
SDS__INVCODE	Invalid type code
SDS__INVDIMS	Invalid dimensions
SDS__NOTSTRUCT	Parent is not a structure
SDS__EXTERN	Parent is external

Support: Jeremy Bailey, AAO

Version date: 18-Oct-91

A.21 SdsPointer — Get a pointer to the data of a primitive item

Function: Get a pointer to the data of a primitive item

Description: Return a pointer to the data of a primitive item. Also return the length of the item. If the data item is undefined and the object is internal storage for the data will be created.

SdsPointer can only be used with primitive items, not with structures.

If necessary (e.g. if the data originated on a machine with different architecture) the data for the object is converted (in place) from the format stored in the data item to that required for the local machine

If the data pointed to by the pointer is updated by a calling program, the program should then call the function SdsFlush to ensure that the data is updated in the original structure. This is necessary because implementations on some machine architectures may have to use a copy of the data rather than the actual data when returning a pointer.

Language: C

Declaration: void SdsPointer(SdsIdType id, void **data, unsigned long *length, StatusType * SDSCONST status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	id	SdsIdType	Identifier of the primitive item.
<	data	void**	Address of a variable to hold the pointer.
<	length	unsigned long*	Length of the data.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__NOMEM	Insufficient memory for creation
SDS__NOTPRIM	Not a primitive item
SDS__UNDEFINED	Data undefined, and object external

Support: Jeremy Bailey, AAO

Version date: 18-Oct-91

A.22 SdsPut — Write data to an object.

Function: Write data to an object.

Description: Write data into an object. The object may be a primitive item or a structure or structure array.

If the object is a structure or structure array, the data from the the buffer is copied into its primitive components in order of their position in the structure. Alignment adjustments are made as necessary to match the alignment of a C struct equivalent to the SDS structure. In the structure or structure array case the offset parameter is ignored.

If the object is primitive data is transferred starting at the position in the item specified by offset.

If the data was previously undefined memory for the data is allocated at this time.

If the data is too long to fit into the object, it will be truncated.

Language: C

Declaration: void SdsPut(SdsIdType id, unsigned long length, unsigned long offset, void *data, long *status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

`>`	id	SdsIdType	Identifier of the primitive item.
`>`	length	unsigned long	Length in bytes of the buffer containing the data.
`>`	offset	unsigned long	Offset into the data object at which to start writing data. The offset is measured in units of the size of each individual item in the array - e.g. 4 bytes for an `INT` or 8 bytes for a `DOUBLE`. The offset is zero to start at the beginning of the array.
`>`	data	void*	Buffer containing the data.
`!`	status	StatusType*	Modified status. Possible failure codes are:

`SDS__NOMEM`	Insufficient memory for creation
`SDS__NOTPRIM`	Not a primitive item
`SDS__UNDEFINED`	Data undefined, and object external

Support: Jeremy Bailey, AAO

Version date: 18-Oct-91

A.23 SdsPutExtra — Write to the extra information field of an object.

Function: Write to the extra information field of an object.

Description: Write a specified number of bytes to the extra information field of an object. A maximum of 128 bytes may be written to an internal object. It is permissible to write to the extra information field of an external object, but the number of bytes written must not exceed the number originally in the object.

Language: C

Declaration: void SdsPutExtra(SdsIdType id, long nextra, char* extra, StatusType `*` `SDSCONST` status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

`>`	id	SdsIdType	Identifier of the object
`>`	nextra	long	Number of bytes of extra information.
`>`	extra	char*	The extra information to be included. nextra bytes are copied into the object.
`!`	status	StatusType*	Modified status. Possible failure codes are:

`SDS__BADID`	The identifier is invalid.
`SDS__EXTRA`	Too much extra data.

Support: Jeremy Bailey, AAO

Version date: 24-Oct-91

A.24 SdsRename — Change the name of an object.

Function: Change the name of an object.

Description: Specify a new name for an object.

Language: C

Declaration: void SdsRename(SdsIdType id, char* name, StatusType * SDSCONST status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	id	SdsIdType	Identifier of the object to be renamed.
>	name	char*	New name for the object. This should have a maximum length of 16 characters including the terminating null.
!	status	long*	Modified status. Possible failure codes are:

SDS__BADID	The identifier is invalid.
SDS__LONGNAME	The name is too long.

Support: Jeremy Bailey, AAO

Version date: 24-Oct-91

A.25 SdsResize — Change the dimensions of an array.

Function: Change the dimensions of an array.

Description: Change the number and/or size of the dimensions of an array. This operation can be performed on primitive arrays or structure arrays. Note that SDS_RESIZE does not move the data elements in the storage representing the array, so there is no guarantee that after resizing the array the same data will be found at the same array index positions as before resizing, though this will be the case for simple changes such as a change in the last dimension only.

If the size of a primitive array is increased the contents of the extra locations is undefined. Decreasing the size causes the data beyond the new limit to be lost.

If a structure array is extended the new elements created will be empty structures. If a structure array is decreased in size, the lost elements and all their components will be deleted.

Language: C

Declaration: void SdsResize(SdsIdType id, long ndims, unsigned long *dims, StatusType * SDSCONST status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	id	SdsIdType	Identifier of the object to be resized.
>	ndims	long	New number of dimensions.
>	dims	unsigned long*	Array of dimensions.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__BADID	The identifier is invalid.
SDS__NOMEM	Insuficient memory.
SDS__EXTERN	Object is external.
SDS__NOTARR	Object is not an array.
SDS__INVDIMS	Dimensions invalid.

Support: Jeremy Bailey, AAO

Version date: 23-Oct-91

A.26 SdsSize — Find the buffer size needed to export an object

Function: Find the buffer size needed to export an object

Description: Return the size which will be needed for a buffer into which the object can be exported using the SdsExport function.

Language: C

Declaration: void SdsSize(SdsIdType id, unsigned long *bytes, StatusType * SDSCONST status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	id	SdsIdType	Identifier of the object.
<	bytes	unsigned long*	Size in bytes of required buffer.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__BADID	The identifier is invalid.
SDS__EXTERN	Object is external.

Support: Jeremy Bailey, AAO

Version date: 18-Oct-91

A.27 SdsSizeDefined — Find the buffer size needed to export using SdsExportDefined

Function: Find the buffer size needed to export using SdsExportDefined

Description: Return the size which will be needed for a buffer into which the object can be exported using the SdsExportDefined function.

Language: C

Declaration: void SdsSizeDefined(SdsIdType id, unsigned long *bytes, StatusType * SDSCONST status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	id	SdsIdType	Identifier of the object.
<	bytes	unsigned long*	Size in bytes of required buffer.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__BADID	The identifier is invalid.
SDS__EXTERN	Object is external.

Support: Jeremy Bailey, AAO

Version date: 13-Jul-98

B SDS Utility Function Descriptions

B.1 SdsFillArray — Fill out the contents of a structured array.

Function: Fill out the contents of a structured array.

Description: This routine will fill out an array of structures item with the copies of a specified struture.

Language: C

Declaration: void SdsFillArray(SdsIdType array_id, SdsIdType elem_id, StatusType *status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

`>`	array_id	SdsIdType	The identifier to the array of structures object to be filled
`>`	elem_id	SdsIdType	The identifier to the object, copies of which are to be put into the array of structures.
`!`	status	long*	Modified status. SdsFillArray calls a large number of `SDS` routines so will return error status values if an error occurs in any of these routines.

Bugs: Due to a missing Sds routine, the resulting structure may have one level too deep.

Prior requirements: None.

Support: Tony Farrell, AAO

Version date: 226-Sep-96

B.2 SdsFindByPath — Accesses a structued Sds item using a path name to the item.

Function: Accesses a structued Sds item using a path name to the item.

Description: This function is passed the id of an Sds structure and a name describing an element in that sturcture using a dot separated format. It returns the id of the element. For example, if we have a structure of the form

```
Version               Float
FibreCentDist         Float
    PlateArray            Struct
       Bundle              Struct
         Fibres            Struct
             xposition             Float
             yposition             Float
             transmission          Float
             bias                  Float
             broken                Short
```

Then the name "PlateArray.BundleArray.Fibres.xposition is a valid name. In addition, structure array elements can be specified using a specifiction like

```
item1[2]
item2[2,3]
```

Where item1 is a one dimensional array and item2 is a two dimensional array.

Note that the use of this routine requires that Sds names not use period or square brackets in their names. This is not enforced any where so must be done by convention.

Language: C

Call:
(void) = SdsFindByPath (parent_id,name,id,status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

`>`	parent_id	SdsIdType	Identified of the structure
`>`	name	char `*`	Structured name of the item to find
`<`	id	SdsIdType `*`	Identifier to the component
`!`	status	long*	Modified status. Possible failure codes are:

`SDS__NOMEM`	Insufficent memory for creation
`SDS__NOTSTRUCT`	Parent id or subsequent non-terminating item is not a structure
`SDS__NOITEM`	No item with the specified name.
`SDS__NOTARRAY`	Attempt to access structure array element in an item which is not a structure array.

Include files: sds.h

Support: Tony Farrell, `AAO`

B.3 SdsList — List contents of an `SDS` object

Function: List contents of an `SDS` object

Description: A listing of the contents of an `SDS` object is generated on standard output. The listing consists of the name type, dimensions and value of each object in the structure. The hierarchical structure is indicated by indenting the listing for components at each level.

For array objects only the values of the first few components are listed so that the listing for each object fits on a single line.

Language: C

Declaration: void SdsList(SdsIdType id, StatusType *status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

>	id	SdsIdType	The identifier to the object to be listed
!	status	long*	Modified status. SdsList calls a large number of SDS routines so will return error status values if an error occurs in any of these routines.

Prior requirements: None.

Support: Jeremy Bailey, AAO

Version date: 29-Apr-96

B.4 SdsRead — Read an SDS object from a file

Function: Read an SDS object from a file

Description: Read an SDS object from a file previously written by SdsWrite. An identifier to an external object is returned. If an internal version of the object is required it can be created using SdsCopy.

Language: C

Declaration: void SdsRead(char *filename, SdsIdType *id, StatusType *status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	filename	char*	The name of the file from which the object will be read.
<	id	SdsIdType*	The identifier to the external object.
!	status	long*	Modified status. Possible failure codes are:

SDS__NOTSDS	Not a valid SDS object
SDS__NOMEM	Insufficient memory
SDS__FOPEN	Error opening the file
SDS__FREAD	Error reading the file

Support: Jeremy Bailey, AAO

Version date: 4-Feb-92

B.5 SdsReadFree — Free Buffer allocated by SdsRead

Function: Free Buffer allocated by SdsRead

Description: SdsRead allocates a block of memory to hold the external object read in. This memory can be released when the object is no longer required by calling SdsReadFree (note that it is not possible to SdsDelete an external object).

If SdsReadFree is given an idientifier which was not produced by a call to SdsRead it will do nothing.

Language: C

Declaration: void SdsReadFree(SdsIdType id, StatusType *status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

`<`	id	SdsIdType*	The identifier to the external object.
`!`	status	long*	Modified status. Possible failure codes are:

`SDS__NOTSDS`	Not a valid `SDS` object
`SDS__NOMEM`	Insufficient memory

Support: Jeremy Bailey, AAO

Version date: 28-Aug-95

B.6 SdsTypeToString — Given an Sds Type Code, return a pointer to a string.

Function: Given an Sds Type Code, return a pointer to a string.

Description: Just simply looks up the code and returns a string pointer describing the type referred to by the Sds Code.

If code is invalid, then will return "`SDS_INVALID`" or "invalid type" depending on mode.

Language: C

Declaration: const char *SdsTypeToStr(code,mode)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

`>`	SdsCodeType	code	The Sds code.
`>`	mode	mode	`IF` true,then return things like "`SDS_STRUCT`". If false, then return things link "Int", "Struct" etc.
`!`	status	StatusType*	Modified status.

Support: Tony Farrell, AAO

Version date: 18-Apr-97

B.7 SdsWrite — Write an `SDS` object to a file

Function: Write an `SDS` object to a file

Description: Given an identifier to an internal `SDS` object, write it to a file. The file can be read back using SdsRead.

Language: C

Declaration: void SdsWrite(SdsIdType id, char *filename, StatusType *status)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

>	id	SdsIdType	The identifier to the object to be output
>	filename	char*	The name of the file into which the object will be written
!	status	long*	Modified status. Possible failure codes are:

SDS__BADID	The identifier is invalid
SDS__EXTERN	The object is external
SDS__NOMEM	Insufficient memory for output buffer
SDS__FOPEN	Error opening the file
SDS__FWRITE	Error writing the file

Prior requirements: None.

Support: Jeremy Bailey, AAO

Version date: 31-Jan-92

B.8 SdsCompiler — Compile a C structure definition to create an SDS structure.

Function: Compile a C structure definition to create an SDS structure.

Description: When given a string containing a valid C structure definition, this routine will create a Sds version of the structure and return its id. This routine is just a run time hook into the “sdsc” program.

Note, that this routines generated by lex and yacc and hence uses any external names in the code generated by those programs.

Declaration: void SdsCompiler(char *string, int messages, int intas32bit, SdsIdType *id, StatusType *status);

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

>	string	char *	A string containing a C structure declaration
>	messages	int	If true, then in addition to setting status, messages will be written to the stderr when parsing error occur. This may help in debugging as only one status value can be returned. There are two possible values

SDS_COMP_MESS_ERROR	Output messages when status would be set.
SDS_COMP_MESS_WARN	Inaddition to error messages, output warnings.

>	intas32bit	int	Set true to treat ambiguous int declarations as 32 bit ints. Clear to treat them as short ints.
<	id	SdsIdType *	The id of the created structure.
!	status	StatusType *	Modified status.

Possible failure codes are:

SDS__INVPRIMTYPE	Invalid primitive type code.
SDS__INVSTRUCTDEF	Invalid structure definition.
SDS__SYNTAX	Parser syntax error.
SDS__INVSTRUCTURE	Invalid structure.
SDS__INVTYPETYPE	Invalid typedef type.
SDS__STRUCTMULTDEF	Multiply defined structure.
SDS__INVINT	Invalid integer.
SDS__INVTYPE	Invalid type.
SDS__INVINPUT	Invalid input.
SDS__STRUCTARRAY	Array of structures.
SDS__MAXDIMS	Exceeded maximum number of dimensions.
SDS__NOINPUT	String was empty.

Prior requirements: None

Support: Tony Farrell, AAO

Version date: 17-Jun-92

C ARG Function Descriptions

C.1 ArgCvt — Convert from one scaler SDS type to Another.

Function: Convert from one scaler SDS type to Another.

Description: This routine meets requirements for a general type conversion between Sds scaler types. There are three classes of scalars

1. Signed Integers
2. Unsigned Integers
3. Real (floating point) values

Within each class, Sds can represent various types. For example- Sds supports SDS_BYTE, SDS_SHORT SDS_INT and SDS_I64 versions of the Signed integer class. The difference is the number of bytes required for each one.

This routine does a three part conversion-

1. Convert the source type to the largest type of the same class (char, short -> long int) (unsigned char, unsigned short -> unsigned long int) (float -> double)
2. Convert the value above to the largest type of the destination class.
3. Convert the value in 2 to the actual destination type.

Range errors are possible during the conversions.

The source/destination for a conversion can be the address of a value of the appropiate type or it may be an Sds item.

When specifing the address of the value, you must specify the type. (SDS_CHAR, SDS_INT, SDS_UINT etc. (Not SDS_STRUCT)). Additional type codes - ARG_STRING/ARG_STRING2 - can be speicifed indicating the source or destination is a null terminated string while should/will contain a representation of the number. (If both source and destination are strings, then a simple string copy is done.) A ARG_STRING2 type differs in only when the source is a FLOAT or DOUBLE value. When ARG_STRING is used, the maximum number of decimal digits is retrieved whilst when ARG_STRING2 is used, 6 is used for FLOAT and 10 for double. (Note that a type of SDS_CHAR represents a single character, not a string of characters)

To specify an Sds item as the source/destination, supply the address of the Sds id of the item in the appropiate address argument. Suppy ARG_SDS as the corresponding type code. The Sds id must describe a Scaler item, expect if it is a one dimensional character array. In this case, it is considered a character string. Source strings must be null terminated.

Invalid conversions result in status being set to ARG__CNVERR and an error being reported using ErsRep.

The ranges of integer types are determined by the range acceptable to SDS. The ranges of real types are determined by the archecture on which the machine is running.

Types of SDS_INT and SDS_UINT indicate the relevant item is a long int (which may be 32 or 64 bits, depending upon the machine and compiler being used). Note that if the machine does not support 64 bits integers then 64bit values with the high 32bits set to non-zero values cannot be handled - an error is returned.

Language: C

Call:

(Void) = ArgCvt (SrcAddr, SrcType, DstType, DstAddr, DstLen, status)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

>	SrcAddr	void *	Address of the source data or of an Sds item id.
>	SrcType	SdsCodeType	Type of the source data.
>	DstType	SdsCodeType	Type of the destination data.
>	DstAddr	void *	Address of the destination or of an Sds item id.
>	DstLen	int	Length of the destination in bytes. If DstType is ARG_SDS, then this is ignored.
!	status	StatusType *	Modified status.

Include files: Arg.h

External functions used:

ErsRep	Ers	Report an error.
ErsSPrintf	Ers	Format a string into a buffer.
strtol	CRTL	Convert a decimal string to a long.
strtoul	CRTL	Convert a decimal string to an unsigned long.
strtod	CRTL	Convert a decimal string to a double.
strncpy	CRTL	Copy one string to another
strlen	CRTL	Get the length of a string.

External values used: none

Prior requirements: none

Support: Tony Farrell, AAO

C.2 ArgDelete — Delete an argument structure

Function: Delete an argument structure

Description: Delete an argument structure and free its identifier

Language: C

Declaration: void ArgDelete(SdsIdType id, const StatusType * status)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

>	id	SdsIdType	Identifier to the structure to be deleted.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__BADID	Invalid Identifier

Support: Jeremy Bailey, AAO

Version date: 7-Apr-92

C.3 ArgFind — Call SdsFind, but report any error using ErsRep.

Function: Call SdsFind, but report any error using ErsRep.

Description: This routine simply calls SdsFind, but if SdsFind returns a bad status, then this routine reports the name of the item SdsFind was trying to find.

Language: C

Declaration: void ArgFind(SdsIdType parent_id, char *name, SdsIdType *id, StatusType * status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	parent_id	SdsIdType	Identifier of the structure.
>	name	char*	Name of the component to be found.
<	id	SdsIdType*	Identifier to the component.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__NOMEM	Insufficient memory for creation
SDS__NOTSTRUCT	parent_id not a structure
SDS__NOITEM	No item with that name

Support: Tony Farrell, AAO

See Also: SdsFind().

Version date: 14-Apr-98

C.4 ArgGetString — Get a character string item from an argument structure

Function: Get a character string item from an argument structure

Description: A character string item is read from a named component within the specified argument structure.

Language: C

Declaration: void ArgGetString(SdsIdType id, const char *name, long len,char *value, const StatusType * status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	id	SdsIdType	Identifier to the structure.
>	name	char*	Name of the component to be read from.
>	len	long	Length of buffer to receive string.
<	value	char*	Character string buffer to read into.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__NOMEM	Insufficient memory for creation
SDS__BADID	Invalid Identifier
ARG__NOTSTRING	Item is not a string

Support: Jeremy Bailey, AAO

Version date: 28-Mar-92

C.5 ArgGetc — Get a character item from an argument structure

Function: Get a character item from an argument structure

Description: A character item is read from a named component within the specified argument structure. The item is converted to character type if necessary.

Language: C

Declaration: void ArgGetc(SdsIdType id, const char *name, char *value, const StatusType * status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

`>`	id	SdsIdType	Identifier to the structure.
`>`	name	char*	Name of the component to be read from.
`>`	value	char*	Character variable to read into.
`!`	status	StatusType*	Modified status. Possible failure codes are:

`SDS__NOMEM`	Insufficient memory for creation
`SDS__BADID`	Invalid Identifier
`ARG__NOTSCALAR`	Item is not a scalar
`ARG__CNVERR`	Type conversion error

Support: Jeremy Bailey, AAO

Version date: 27-Mar-92

C.6 ArgGetd — Get a double floating point item from an argument structure

Function: Get a double floating point item from an argument structure

Description: A double floating point item is read from a named component within the specified argument structure. The item is converted to double type if necessary.

Language: C

Declaration: void ArgGetd(SdsIdType id, const char *name, double *value, const StatusType * status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

`>`	id	SdsIdType	Identifier to the structure.
`>`	name	char*	Name of the component to be read from.
`>`	value	double*	Double variable to read into.
`!`	status	StatusType*	Modified status. Possible failure codes are:

`SDS__NOMEM`	Insufficient memory for creation
`SDS__BADID`	Invalid Identifier
`ARG__NOTSCALAR`	Item is not a scalar
`ARG__CNVERR`	Type conversion error

Support: Jeremy Bailey, AAO

Version date: 27-Mar-92

C.7 ArgGetf — Get a floating point item from an argument structure

Function: Get a floating point item from an argument structure

Description: A floating point item is read from a named component within the specified argument structure. The item is converted to float type if necessary.

Language: C

Declaration: void ArgGetf(SdsIdType id, const char *name, float *value, const StatusType * status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

>	id	SdsIdType	Identifier to the structure.
>	name	char*	Name of the component to be read from.
>	value	float*	Float variable to read into.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__NOMEM	Insufficient memory for creation
SDS__BADID	Invalid Identifier
ARG__NOTSCALAR	Item is not a scalar
ARG__CNVERR	Type conversion error

Support: Jeremy Bailey, AAO

Version date: 27-Mar-92

C.8 ArgGeti — Get an integer item from an argument structure

Function: Get an integer item from an argument structure

Description: A long integer integer item is read from a named component within the specified argument structure. The item is converted to long integer type if necessary.

Language: C

Declaration: void ArgGeti(SdsIdType id, const char *name, long *value, const StatusType * status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

>	id	SdsIdType	Identifier to the structure.
>	name	char*	Name of the component to be read from.
>	value	long*	Long variable to read into.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__NOMEM	Insufficient memory for creation
SDS__BADID	Invalid Identifier
ARG__NOTSCALAR	Item is not a scalar
ARG__CNVERR	Type conversion error

Support: Jeremy Bailey, AAO

Version date: 27-Mar-92

C.9 ArgGeti64 — Get a 64 bit integer item from an argument structure

Function: Get a 64 bit integer item from an argument structure

Description: A 64 bit integer integer item is read from a named component within the specified argument structure. The item is converted to 64 bit integer type if necessary.

Language: C

Declaration: void ArgGeti64(SdsIdType id, const char *name, `INT64` *value, const StatusType `*` status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

>	id	SdsIdType	Identifier to the structure.
>	name	char*	Name of the component to be read from.
>	value	`INT64*`	Long variable to read into.
!	status	StatusType*	Modified status. Possible failure codes are:

`SDS__NOMEM`	Insufficient memory for creation
`SDS__BADID`	Invalid Identifier
`ARG__NOTSCALAR`	Item is not a scalar
`ARG__CNVERR`	Type conversion error

Support: Jeremy Bailey, AAO

Version date: 27-Mar-92

C.10 ArgGets — Get a short integer item from an argument structure

Function: Get a short integer item from an argument structure

Description: A short integer item is read from a named component within the specified argument structure. The item is converted to short type if necessary.

Language: C

Declaration: void ArgGets(SdsIdType id, const char *name, short *value, const StatusType `*` status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

>	id	SdsIdType	Identifier to the structure.
>	name	char*	Name of the component to be read from.
>	value	short*	Short variable to read into.
!	status	StatusType*	Modified status. Possible failure codes are:

`SDS__NOMEM`	Insufficient memory for creation
`SDS__BADID`	Invalid Identifier
`ARG__NOTSCALAR`	Item is not a scalar
`ARG__CNVERR`	Type conversion error

Support: Jeremy Bailey, AAO

Version date: 27-Mar-92

C.11 ArgGetu — Get an unsigned integer item from an argument structure

Function: Get an unsigned integer item from an argument structure

Description: An unsigned long integer item is read from a named component within the specified argument structure. The item is converted to unsigned long type if necessary.

Language: C

Declaration: void ArgGetu(SdsIdType id, const char *name, unsigned long *value, const StatusType * status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	id	SdsIdType	Identifier to the structure.
>	name	char*	Name of the component to be read from.
>	value	unsigned long*	Unsigned long variable to read into.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__NOMEM	Insufficient memory for creation
SDS__BADID	Invalid Identifier
ARG__NOTSCALAR	Item is not a scalar
ARG__CNVERR	Type conversion error

Support: Jeremy Bailey, AAO

Version date: 27-Mar-92

C.12 ArgGetu64 — Get an unsigned 64bit integer item from an argument structure

Function: Get an unsigned 64bit integer item from an argument structure

Description: An unsigned 64bit integer item is read from a named component within the specified argument structure. The item is converted to unsigned 64bit type if necessary.

Language: C

Declaration: void ArgGetu64(SdsIdType id, const char *name, UINT64 *value, const StatusType * status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	id	SdsIdType	Identifier to the structure.
>	name	char*	Name of the component to be read from.
>	value	UINT64*	Unsigned 64bit variable to read into.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__NOMEM	Insufficient memory for creation
SDS__BADID	Invalid Identifier
ARG__NOTSCALAR	Item is not a scalar
ARG__CNVERR	Type conversion error

Support: Tony Farrell, AAO

Version date: 10-Aug-1995

C.13 ArgGetus — Get an unsigned short integer item from an argument structure

Function: Get an unsigned short integer item from an argument structure

Description: An unsigned short integer item is read from a named component within the specified argument structure. The item is converted to unsigned short type if necessary.

Language: C

Declaration: void ArgGetus(SdsIdType id, const char *name, unsigned short *value, const StatusType * status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	id	SdsIdType	Identifier to the structure.
>	name	char*	Name of the component to be read from.
>	value	unsigned short*	Short variable to read into.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__NOMEM	Insufficient memory for creation
SDS__BADID	Invalid Identifier
ARG__NOTSCALAR	Item is not a scalar
ARG__CNVERR	Type conversion error

Support: Jeremy Bailey, AAO

Version date: 27-Mar-92

C.14 ArgLook — Look at the contents of a string

Function: Look at the contents of a string and determined if it can be represented as a number.

Description: A common requirement in user interfaces is to convert a string containing a number to the machine representation of that number. This routine determines if this is possible and if so, what type the string can be converted to.

1. If the string has the format - [+-]nnn, where nnn means any number of decimal digits, this routine beleives it can be be represented as an integer.

2. If the string has the format - [+-]nnn[.nnn][e—E[+/-]nnn], where nnn means nay number of decimal digites, this routine beleives it can be represented as a real.

The actual ability to represent the number in a given machine type is dependent on the size of the number.

If MinFlag is flase and the string has the format of 1, then DstType is set to `SDS_INT`, except if USFlag is set true, In which case DstType is set to `SDS_UINT` (unless the number is negative).

If MinFlag is false and the string has the format of 2, then DstType is set to `SDS_DOUBLE`.

If MinFlag is true, then the routine attempts to determine the smallest size of the required type which can be used to represent the number. If base type (`INT` or `REAL`) is determined as per when MinFlag is false. The system then tries to convert the value to a number of that type using ArgCvt. If this is successfull, then it looks at the resulting values and determines the smallest type which can be used to represet the number.

If the string is not a valid number (or when MinFlag is true, cannot be represented in the largest type) then DstType is set to `ARG_STRING`.

Any precedding white space in the string is ignored, but trailing white space is not allowed.

Language: C

Call:

(Void) = ArgLook (SrcAddr, USFlag, MinFlag, DstType , status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

`>`	SrcAddr	char *	Address of the source data.
`>`	USFlag	int	If true, integers are unsigned unless they include a minus sign. If false, integers are always signed.
`>`	MinFlag	int	If true, then find the smallest type of the appropiate class which can represent this time. E.g., use float instead of double if possible. This is a more expensive operation as ArgLook must invoke ArgCvt to do a conversion in order to determine this.
`<`	DstType	SdsCodeType *	Type of the destination.
`!`	status	Long int *	Modified status.

Include files: Arg.h

External functions used:

ErsPush	Ers	Increment error context.
ErsPop	Ers	Decrement error context.
ErsAnnul	Ers	Annull error messages.
ArgCvt	Arg	Convert a value from one type to another.
isspace	`CRTL`	Is a character a which space character.
isdigit	`CRTL`	Is a character a digit.

External values used: none

Prior requirements: none

Support: Tony Farrell, `AAO`

C.15 ArgNew — Create a new argument structure

Function: Create a new argument structure

Description: Creates a new structure to hold arguments and return an identifier to it

Language: C

Declaration: void ArgNew(SdsIdType *id, long *status)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

<	id	SdsIdType*	Identifier to the created structure
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__NOMEM	Insufficient memory for creation

Support: Jeremy Bailey, AAO

Version date: 26-Mar-92

C.16 ArgPutString — Put a character string item into an argument structure

Function: Put a character string item into an argument structure

Description: A character string item is written into a named component within the specified argument structure. The component is created if it does not currently exist.

Language: C

Declaration: void ArgPutString(SdsIdType id, char *name, char *value, const StatusType * status)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

>	id	SdsIdType	Identifier to the structure.
>	name	char*	Name of the component to be written to.
>	value	char*	Null terminated string to be written.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__NOMEM	Insufficient memory for creation
SDS__BADID	Invalid Identifier
ARG__NOTSTRING	Item is not a string

Support: Jeremy Bailey, AAO

Version date: 28-Mar-92

C.17 ArgPutc — Put a character item into an argument structure

Function: Put a character item into an argument structure

Description: A character item is written into a named component within the specified argument structure. The component is created if it does not currently exist.

Language: C

Declaration: void ArgPutc(SdsIdType id, const char *name, char value, const StatusType * status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	id	SdsIdType	Identifier to the structure.
>	name	char*	Name of the component to be written to.
>	value	char	Character value to be written.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__NOMEM	Insufficient memory for creation
SDS__BADID	Invalid Identifier
ARG__NOTSCALAR	Item is not a scalar
ARG__CNVERR	Type conversion error

Support: Jeremy Bailey, AAO

Version date: 26-Mar-92

C.18 ArgPutd — Put a double floating point item into an argument structure

Function: Put a double floating point item into an argument structure

Description: A double item is written into a named component within the specified argument structure. The component is created if it does not currently exist.

Language: C

Declaration: void ArgPutd(SdsIdType id, const char *name, double value, const StatusType * status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	id	SdsIdType	Identifier to the structure.
>	name	char*	Name of the component to be written to.
>	value	double	Floating point value to be written.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__NOMEM	Insufficient memory for creation
SDS__BADID	Invalid Identifier
ARG__NOTSCALAR	Item is not a scalar
ARG__CNVERR	Type conversion error

Support: Jeremy Bailey, AAO

Version date: 27-Mar-92

C.19 ArgPutf — Put a floating point item into an argument structure

Function: Put a floating point item into an argument structure

Description: A floating point item is written into a named component within the specified argument structure. The component is created if it does not currently exist.

Language: C

Declaration: void ArgPutf(SdsIdType id, const char *name, float value, const StatusType * status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

`>`	id	SdsIdType	Identifier to the structure.
`>`	name	char*	Name of the component to be written to.
`>`	value	float	Floating point value to be written.
`!`	status	StatusType*	Modified status. Possible failure codes are:

`SDS__NOMEM`	Insufficient memory for creation
`SDS__BADID`	Invalid Identifier
`ARG__NOTSCALAR`	Item is not a scalar
`ARG__CNVERR`	Type conversion error

Support: Jeremy Bailey, AAO

Version date: 27-Mar-92

C.20 ArgPuti — Put a integer item into an argument structure

Function: Put a integer item into an argument structure

Description: An integer item is written into a named component within the specified argument structure. The component is created if it does not currently exist.

Note, when this call has to create the item, it always creates 32bit integer `SDS` items, regardless of the size of long int on the machine. Use ArgPuti64 to create 64bit integer items.

Language: C

Declaration: void ArgPuti(SdsIdType id, const char *name, long value, const StatusType * status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

`>`	id	SdsIdType	Identifier to the structure.
`>`	name	char*	Name of the component to be written to.
`>`	value	long	Integer value to be written.
`!`	status	StatusType*	Modified status. Possible failure codes are:

`SDS__NOMEM`	Insufficient memory for creation
`SDS__BADID`	Invalid Identifier
`ARG__NOTSCALAR`	Item is not a scalar
`ARG__CNVERR`	Type conversion error

Support: Jeremy Bailey, AAO

Version date: 27-Mar-92

C.21 ArgPuti64 — Put a 64 bit integer item into an argument structure

Function: Put a 64 bit integer item into an argument structure

Description: A 64 bit integer item is written into a named component within the specified argument structure. The component is created if it does not currently exist.

Language: C

Declaration: void ArgPuti64(SdsIdType id, const char *name, `INT64` value, const StatusType * status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

`>`	id	SdsIdType	Identifier to the structure.
`>`	name	char*	Name of the component to be written to.
`>`	value	`INT64`	64 bit integer value to be written.
`!`	status	StatusType*	Modified status. Possible failure codes are:

`SDS__NOMEM`	Insufficient memory for creation
`SDS__BADID`	Invalid Identifier
`ARG__NOTSCALAR`	Item is not a scalar
`ARG__CNVERR`	Type conversion error

Support: Tony Farrell, AAO

Version date: 10-Aug-95

C.22 ArgPuts — Put a short integer item into an argument structure

Function: Put a short integer item into an argument structure

Description: A short integer item is written into a named component within the specified argument structure. The component is created if it does not currently exist.

Language: C

Declaration: void ArgPuts(SdsIdType id, const char *name, short value, const StatusType * status)

Parameters: ("`>`" input, "`!`" modified, "W" workspace, "`<`" output)

`>`	id	SdsIdType	Identifier to the structure.
`>`	name	char*	Name of the component to be written to.
`>`	value	short	Short integer value to be written.
`!`	status	StatusType*	Modified status. Possible failure codes are:

`SDS__NOMEM`	Insufficient memory for creation
`SDS__BADID`	Invalid Identifier
`ARG__NOTSCALAR`	Item is not a scalar
`ARG__CNVERR`	Type conversion error

Support: Jeremy Bailey, AAO

Version date: 27-Mar-92

C.23 ArgPutu — Put an unsigned integer item into an argument structure

Function: Put an unsigned integer item into an argument structure

Description: An unsigned integer item is written into a named component within the specified argument structure. The component is created if it does not currently exist.

Note, when this call has to create the item, it always creates 32bit unsigned integer SDS items, regardless of the size of long int on the machine. Use ArgPutui64 to create 64bit unsigned integer items.

Language: C

Declaration: void ArgPutu(SdsIdType id, const char *name, unsigned long value, const StatusType * status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	id	SdsIdType	Identifier to the structure.
>	name	char*	Name of the component to be written to.
>	value	unsigned long	Unsigned integer value to be written.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__NOMEM	Insufficient memory for creation
SDS__BADID	Invalid Identifier
ARG__NOTSCALAR	Item is not a scalar
ARG__CNVERR	Type conversion error

Support: Jeremy Bailey, AAO

Version date: 27-Mar-92

C.24 ArgPutu64 — Put an unsigned 64 bit integer item into an argument structure

Function: Put an unsigned 64 bit integer item into an argument structure

Description: An unsigned 64 bit integer item is written into a named component within the specified argument structure. The component is created if it does not currently exist.

Language: C

Declaration: void ArgPutu64(SdsIdType id, const char *name, unsigned long value, const StatusType * status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	id	SdsIdType	Identifier to the structure.
>	name	char*	Name of the component to be written to.
>	value	UINT64	Unsigned integer value to be written.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__NOMEM	Insufficient memory for creation
SDS__BADID	Invalid Identifier
ARG__NOTSCALAR	Item is not a scalar
ARG__CNVERR	Type conversion error

Support: Tony Farrell, AAO

Version date: 10-Aug-95

C.25 ArgPutus — Put an unsigned short integer item into an argument structure

Function: Put an unsigned short integer item into an argument structure

Description: An unsigned short item is written into a named component within the specified argument structure. The component is created if it does not currently exist.

Language: C

Declaration: void ArgPutus(SdsIdType id, const char *name, unsigned short value, const StatusType * status)

Parameters: (">" input, "!" modified, "W" workspace, "<" output)

>	id	SdsIdType	Identifier to the structure.
>	name	char*	Name of the component to be written to.
>	value	unsigned short	Unsigned short value to be written.
!	status	StatusType*	Modified status. Possible failure codes are:

SDS__NOMEM	Insufficient memory for creation
SDS__BADID	Invalid Identifier
ARG__NOTSCALAR	Item is not a scalar
ARG__CNVERR	Type conversion error

Support: Jeremy Bailey, AAO

Version date: 27-Mar-92

C.26 ArgSdsList — List an Sds structure calling a user supplied callback.

Function: List an Sds structure calling a user supplied callback.

Description: This routine does basically the same job as SdsList, but invokes a user supplied callback routine to do the actual output. This allows the output to be directed anywhere required by the user, not just to stdout as in SdsList.

The output routine is called for each line to be output. Note that although close to the output of SdsList, this output of this routine is not exactly the same.

Language: C

Declaration: void ArgSdsList(id, buflen, buffer, func, client_data, status)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

>	id	SdsIdType	Identifier to the structure.
>	buflen	unsigned int	Length of buffer. If zero, then malloc a buffer of 400 bytes. If lines exceed this length, then they are truncated correctly before calling the output routine.
>	buffer	char *	A buffer to be used by this routine to print the output into. It is passed directly to the output function. Since the length of each line can be quite long (particularly with arrays) we allow the user to specify how much of a line he is interested in. For example, if outputing to a terminal, you might only specify an 80 byte buffer. If not supplied, we will malloc buflen bytes.
>	func	ArgListFunctType	A function to called to output each line. It is passed the client data item, the address of the buffer and a modified status item.
>	client_data	void *	Passed directly to func.
!	status	StatusType*	Modified status.

Support: Tony Farrell, AAO

Version date: 17-Apr-97

C.27 ArgToString — Take an Sds structure and write it to a string.

Function: Take an Sds structure and write it to a string.

Description: The structure is examined recursively. It scraler or string item found is written to the output string, with a space separating each item. No structure is maintined to the data.

The normal use of the function is to convert what is expected to be simple structures in to something suitable for output to the user.

Language: C

Call:
(Void) = ArgToString (id,maxlen,length,string,status)

Parameters: (“>” input, “!” modified, “W” workspace, “<” output)

>	id	SdsIdType	Identifier to the structure to be deleted.
>	maxlen	Int	Length of buffer to receive string.
!	length	Int *	Set to zero on entry. On exit, will contain the actual length of the string.
<	string	Char *	Character strin g buffer to write to.
!	status	Long int *	Modified status.

Include files: Arg.h

External functions used:

External values used: none

Prior requirements: none

Support: Tony Farrell, AAO

D SDS Fortran Subroutine Interface

D.1 SDS subroutines

SDS_ACCESS Return an identifier to an external object SDS_ACCESS

Description: Make an external object (one exported by SDS_EXPORT) accessible to SDS by returning an identifier to it.

Any SDS operations which do not change the structure of the object may be performed on an external object. These include navigation operations (SDS_FIND, SDS_INDEX, SDS_CELL), data access operations (SDS_GET, SDS_PUT, SDS_POINTER) and inquiry operations (SDS_INFO).

Operations which are not permitted on an external object are those which add or remove components (SDS_NEW, SDS_DELETE), or write operations (SDS_PUT or SDS_POINTER) to data items which are currently undefined.

Unlike SDS_IMPORT, SDS_ACCESS does not make a copy of the object. The object is accessed in place in the original buffer.

Invocation: `CALL SDS_ACCESS( DATA, ID, STATUS )`

Arguments:

DATA(*) = BYTE (Given)
The buffer containing the object to be accessed.

ID = INTEGER (Returned)
The identifier of the external object.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__NOTSDS Not a valid SDS object.
- SDS__NOMEM Insufficient memory.

SDS_CELL Find component of a structure array SDS_CELL

Description: Given the indices to a component of a structure array, return an identifier to it.

Invocation: `CALL SDS_CELL( ARRAY_ID, NINDICES, INDICES, ID, STATUS )`

Arguments:

ARRAY_ID = INTEGER (Given)
The identifier of the structure array

NINDICES = INTEGER (Given)
Number if indices supplied in the array INDICES. This should be the same as the number of dimensions of the array

INDICES (NINDICES) = INTEGER (Given)
An array of length NINDICES containing the indices to the component of the structure array

ID = INTEGER (Returned)
The identifier of the new internal structure.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID The identifier is invalid.
- SDS__NOMEM Insufficient memory.
- SDS__NOTARRAY Not a strcture array.
- SDS__INDEXERR Indices invalid.

SDS_COPY Make a copy of an object SDS_COPY

Description: Make a copy of an object and return an identifier to the copy. The copy is a new top level object, the original object is unchanged by the operation.

The object being copied can be either external or internal. The copy is always an internal object.

Invocation: `CALL SDS_COPY( ID, COPY_ID, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object to be copied.

COPY_ID = INTEGER (Returned)
The identifier of the new internal structure.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID The identifier is invalid.
- SDS__NOMEM Insufficient memory.

SDS_DELETE Delete an object SDS_DELETE

Description: Delete an object, freeing any memory associated with it. Subsequent attempts to access the object through any identifier associated with it will return the SDS__BADID status. A structure array element cannot be deleted. An attempt to do so will result in the SDS__ILLDEL status.

Deleting an object does not free the memory associated with the identifier referencing it. This memory can be freed with the SDS_FREE_ID function.

Invocation: `CALL SDS_DELETE( ID, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object to be deleted.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID The identifier is invalid.
- SDS__EXTERN Object is external.
- SDS__ILLDEL Object cannot be deleted.

SDS_EXPORT Export an object SDS_EXPORT

Description: Export an object into an external buffer.

Once exported an object can be moved around in memory, written to a file etc., and subsequently returned to the SDS system either by using SDS_IMPORT to import it back into the system, or SDS_ACCESS, to access it as an external object.

The original internal version of the object continues to exist, in addition to the external copy. All identifiers to the object continue to refer to the original internal copy.

The length of the buffer required for SDS_EXPORT can be determined by a call to SDS_SIZE.

Invocation: `CALL SDS_EXPORT( ID, LENGTH, DATA, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object to be exported.

LENGTH = INTEGER (Given)
Size in bytes of the buffer.

DATA (LENGTH) = BYTE (Returned)
The buffer into which the object will be exported

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID The identifier is invalid.
- SDS__TOOLONG The object is too large for the buffer.
- SDS__EXTERN Object is external.

SDS_EXTRACT Extract an object from a structure SDS_EXTRACT

Description: Extract an object from a structure. The extracted object becomes a new independent top level object. The object is deleted from the original structure.

The identifier must not be that of a structure array component.

If the identifier is already that of a top level object, then the function does nothing.

Invocation: `CALL SDS_EXTRACT( ID, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object to be extracted

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID The identifier is invalid.
- SDS__EXTERN Object is external.
- SDS__ILLDEL Object cannot be extracted.

SDS_FIND Find a structure component by name SDS_FIND

Description: Given the name of a component in a structure, return an identifier to the component.

Invocation: `CALL SDS_FIND( PARENT_ID, NAME, ID, STATUS )`

Arguments:

PARENT_ID = INTEGER (Given)
Identifier of the structure

NAME = CHAR (Given)
Name of the component to be found

ID = INTEGER (Returned)
Identifier to the component

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__NOMEM Insufficient memory.
- SDS__NOTSTRUCT PARENT_ID not a structure.
- SDS__NOITEM No item with that name.

SDS_FIND_BY_PATH Accesses a structued Sds item using a path name to the item. SDS_FIND_BY_PATH

Description: This function is passed the id of an Sds structure and a name describing an element in that sturcture using a dot separated format. It returns the id of the element. See the C routine SdsFindByPath for details of the naming format.

Invocation: `CALL SDS_FIND_BY_PATH( PARENT_ID, NAME, ID, STATUS )`

Arguments:

PARENT_ID = INTEGER (Given)
Identifier of the structure

NAME = CHAR (Given)
Name of the component to be found

ID = INTEGER (Returned)
Identifier to the component

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__NOMEM Insufficient memory.
- SDS__NOTSTRUCT PARENT_ID not a structure.
- SDS__NOITEM No item with that name.

SDS_FLUSH Flush data updated via a pointer SDS_FLUSH

Description: If a primitive data item is accessed via SDS_POINTER, and the data array updated via the returned pointer, then SDS_FLUSH must be called to ensure that the data is updated in the original structure.

This must be done since implementations on some machine architectures may have to use a copy of the data rather than the actual data when returning a pointer.

Invocation: `CALL SDS_FLUSH( ID, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID Invalid Identifier.
- SDS__NOTPRIM Not a primitive item.

SDS_FREE_ID Free an identifer, so that its associated memory may be reused. SDS_FREE_ID

Description: Each identifier allocated by SDS uses memory. To avoid excessive allocation of memory the SDS_FREE_ID function can be used to free the memory assocaited with an identifier that is no longer needed. When this is done, the memory will be re-used by SDS for a subsequent identifer when needed.

Note that if the identifer refers to a top-level structure, you should call SDS_DELETE (for structures created using SDS_NEW.) or SDS_READ_FREE (for structures created using SDS_READ()) before calling SDS_FREE_ID to ensure all memory is recovered.

Invocation: `CALL SDS_FREE_ID( ID, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier to be freeed

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID The identifier is invalid.

SDS_GET Read the data from an object SDS_GET

Description: The object may be a primitive item or a structure or structure array. Read the data from an item into a buffer. If the object is primitive data is transferred starting at the position in the item specified by offset, until the buffer if filled, or the end of the data array is reached.

If the object is a structure or structure array, the data from all its primitive components are copied into the buffer in order of their position in the structure. Alignment adjustments are made as

necessary to match the alignment of an C struct equivalent to the SDS structure. (Since these alignment requirements are machine dependent the actual sequence of bytes returned could be different on different machines). In the structure or structure array case the offset parameter is ignored.

Note that the structures returned from SDS are designed to be used from C, and are not guaranteed to correctly match Fortran structures, in those implementations of Fortran which support a non-standard structure extension.

If necessary (e.g. if the data originated on a machine with different architecture) the data for the object is converted (in place) from the format stored in the data item to that required for the local machine.

Invocation: `CALL SDS_GET( ID, LENGTH, OFFSET, DATA, ACTLEN, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object

LENGTH = INTEGER (Given)
Length in bytes of the buffer supplied to receive the data.

OFFSET = INTEGER (Given)
Offset into the data at which to start reading data. The offset is measured in units of the size of each individual item in the array - e.g. 4 bytes for an INT or 8 bytes for a DOUBLE. The offset is zero to start at the beginning of the array. This parameter is ignored if the object is a structure or structure array.

DATA (LENGTH) = BYTE (Returned)
Buffer to receive the data.

ACTLEN = INTEGER (Returned)
Actual number of bytes transferred.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID Invalid Identifier.
- SDS__UNDEFINED Data undefined.

SDS_GETC Read character data from an object SDS_GETC

Description: Read data from a primitive character object. Data is transferred starting at the position in the item specified by offset, until the buffer if filled, or the end of the data array is reached.

An error will be returned if the item is not a primitive integer object.

If necessary (e.g. if the data originated on a machine with different architecture) the data for the object is converted (in place) from the format stored in the data item to that required for the local machine.

Invocation: `CALL SDS_GETC( ID, LENGTH, OFFSET, DATA, ACTLEN, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object

LENGTH = INTEGER (Given)
Length of the buffer supplied to receive the data.

OFFSET = INTEGER (Given)
Offset into the data at which to start reading data. The offset is zero to start at the beginning of the array.

DATA (LENGTH) = CHARACTER (Returned)
Buffer to receive the data.

ACTLEN = INTEGER (Returned)
Actual number of elements transferred.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID Invalid Identifier.
- SDS__TYPE Object not of character type.
- SDS__UNDEFINED Data undefined.

SDS_GETD Read double precision data from an object SDS_GETD

Description: Read data from a primitive double precision object. Data is transferred starting at the position in the item specified by offset, until the buffer if filled, or the end of the data array is reached.

An error will be returned if the item is not a primitive double precision object.

If necessary (e.g. if the data originated on a machine with different architecture) the data for the object is converted (in place) from the format stored in the data item to that required for the local machine.

Invocation: `CALL SDS_GETD( ID, LENGTH, OFFSET, DATA, ACTLEN, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object

LENGTH = INTEGER (Given)
Length of the buffer supplied to receive the data.

OFFSET = INTEGER (Given)
Offset into the data at which to start reading data. The offset is zero to start at the beginning of the array.

DATA (LENGTH) = DOUBLE (Returned)
Buffer to receive the data.

ACTLEN = INTEGER (Returned)
Actual number of elements transferred.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID Invalid Identifier.
- SDS__TYPE Object not of double type.
- SDS__UNDEFINED Data undefined.

SDS_GETI Read integer data from an object SDS_GETI

Description: Read data from a primitive integer object. Data is transferred starting at the position in the item specified by offset, until the buffer if filled, or the end of the data array is reached.

An error will be returned if the item is not a primitive integer object.

If necessary (e.g. if the data originated on a machine with different architecture) the data for the object is converted (in place) from the format stored in the data item to that required for the local machine.

Invocation: `CALL SDS_GETI( ID, LENGTH, OFFSET, DATA, ACTLEN, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object

LENGTH = INTEGER (Given)
Length of the buffer supplied to receive the data.

OFFSET = INTEGER (Given)
Offset into the data at which to start reading data. The offset is zero to start at the beginning of the array.

DATA (LENGTH) = INTEGER (Returned)
Buffer to receive the data.

ACTLEN = INTEGER (Returned)
Actual number of elements transferred.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID Invalid Identifier.
- SDS__TYPE Object not of integer type.
- SDS__UNDEFINED Data undefined.

SDS_GETL Read logical data from an object SDS_GETL

Description: Read data from a primitive logical object. Data is transferred starting at the position in the item specified by offset, until the buffer if filled, or the end of the data array is reached.

Since the SDS kernel does not support a logical type an integer item is used to represent Fortran logical values. An error will be returned if the item is not a primitive integer object.

If necessary (e.g. if the data originated on a machine with different architecture) the data for the object is converted (in place) from the format stored in the data item to that required for the local machine.

Invocation: `CALL SDS_GETL( ID, LENGTH, OFFSET, DATA, ACTLEN, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object

LENGTH = INTEGER (Given)
Length of the buffer supplied to receive the data.

OFFSET = INTEGER (Given)
Offset into the data at which to start reading data. The offset is zero to start at the beginning of the array.

DATA (LENGTH) = LOGICAL (Returned)
Buffer to receive the data.

ACTLEN = INTEGER (Returned)
Actual number of elements transferred.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID Invalid Identifier.
- SDS__TYPE Object not of logical type.
- SDS__UNDEFINED Data undefined.

SDS_GETR Read real data from an object SDS_GETR

Description: Read data from a primitive real object. Data is transferred starting at the position in the item specified by offset, until the buffer if filled, or the end of the data array is reached.

An error will be returned if the item is not a primitive real object.

If necessary (e.g. if the data originated on a machine with different architecture) the data for the object is converted (in place) from the format stored in the data item to that required for the local machine.

Invocation: `CALL SDS_GETR( ID, LENGTH, OFFSET, DATA, ACTLEN, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object

LENGTH = INTEGER (Given)
Length of the buffer supplied to receive the data.

OFFSET = INTEGER (Given)
Offset into the data at which to start reading data. The offset is zero to start at the beginning of the array.

DATA (LENGTH) = REAL (Returned)
Buffer to receive the data.

ACTLEN = INTEGER (Returned)
Actual number of elements transferred.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID Invalid Identifier.
- SDS__TYPE Object not of real type.
- SDS__UNDEFINED Data undefined.

SDS_GET_EXTRA Read from the extra information field of an object SDS_GET_EXTRA

Description: Read bytes from the extra information field of an object. Bytes are copied until the supplied buffer is filled up or until all bytes in the field are copied.

Invocation: `CALL SDS_GET_EXTRA( ID, LENGTH, EXTRA, ACTLEN, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object

LENGTH = INTEGER (Given)
Length of the buffer supplied to receive the data.

EXTRA = CHAR (Returned)
Buffer to receive the data.

ACTLEN = INTEGER (Returned)
Actual number of bytes transferred.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID Invalid Identifier.

SDS_IMPORT Import an object from an external buffer SDS_IMPORT

Description: Import an object from an external buffer and return an identifier to the internal copy created. The object must have been previously exported using SDS_EXPORT.

The original external version of the structure continues to exist, in addition to the internal copy.

A fully dynamic internal structure is created in which all SDS operations are valid. However, to merely access the data in an object SDS_ACCESS can be used in place of SDS_IMPORT.

Invocation: `CALL SDS_IMPORT( DATA, ID, STATUS )`

Arguments:

DATA(*) = BYTE (Given)
The buffer from which the object will be imported

ID = INTEGER (Returned)
The identifier of the new internal structure.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__NOTSDS Not a valid SDS object.
- SDS__NOMEM Insufficient memory.

SDS_INDEX Find a structure component by position SDS_INDEX

Description: Given the name of a component in a structure, return an identifier to the component.

Invocation: `CALL SDS_INDEX( PARENT_ID, INDEX, ID, STATUS )`

Arguments:

PARENT_ID = INTEGER (Given)
Identifier of the structure

NAME = CHAR (Given)
Name of the component to be found

ID = INTEGER (Returned)
Identifier to the component

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__NOMEM Insufficient memory.
- SDS__NOTSTRUCT PARENT_ID not a structure.
- SDS__NOITEM No item with that name.

SDS_INFO Return information about an object SDS_INFO

Description: Given the identifier to an object, return the name, type code and dimensions of the object.

Invocation: `CALL SDS_INFO( ID, NAME, CODE, NDIMS, DIMS, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object

NAME = CHAR (Returned)
The name of the data object.

CODE = INTEGER (Returned)
The type code for the object.

NDIMS = INTEGER (Returned)
The number of dimensions (if the object is a primitive or structure array)

DIMS (7) = INTEGER (Returned)
The dimensions of the data. An array of size at least 7 should be provided to receive this.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID The identifier is invalid.

SDS_INSERT Insert an existing object into a structure SDS_INSERT

Description: An existing top level object is inserted into a structure.

Invocation: `CALL SDS_INSERT( PARENT_ID, ID, STATUS )`

Arguments:

PARENT_ID = INTEGER (Given)
Identifier of the structure

ID = INTEGER (Returned)
Identifier to the component

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID Invalid identifier.
- SDS__NOTSTRUCT PARENT_ID not a structure.
- SDS__NOTTOP Not a top level object.
- SDS__NOMEM Insufficent memory.
- SDS__EXTERN Object is external.

SDS_NEW Create a new component in a structure SDS_NEW

Description: Creates a new component in an existing internal structure. The new component can be a structure, a structure array, or a primitive. A structure array is specified by means of a type code of SDS_STRUCT and a non-zero number for NDIMS. If the type code is SDS_STRUCT and NDIMS is zero an ordinary structure is created, A primitive type is specified by the appropriate type code.

Invocation: `CALL SDS_NEW( PARENT_ID, NAME, NEXTRA, EXTRA, CODE, NDIMS, DIMS, ID, STATUS )`

Arguments:

PARENT_ID = INTEGER (Given)
The identifier of the structure to which the component is to be added.

NAME = CHAR (Given)
The name of the structure to create. The name should be of maximum length 16 characters including the terminating null.

NEXTRA = INTEGER (Given)
The number of bytes of extra information to be included (maximum 128).

EXTRA = CHAR (Given)
The extra information to be included with the item. NEXTRA characters from here are copied into the structure

CODE = INTEGER (Given)
The type code for the item to be created. If CODE = SDS_STRUCT a structured item will be created. Other codes result in primitive items.

NDIMS = INTEGER (Given)
Number of dimensions for the item. Zero to create a scalar item.

DIMS (NDIMS) = INTEGER (Given)
Array of dimensions for the item. Should be of size at least NDIMS.

ID = INTEGER (Returned)
Identifier of the object to be created.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID Invalid Identifier.
- SDS__NOMEM Insufficient memory.
- SDS__LONGNAME Name it too long.
- SDS__EXTRA Too much extra data.
- SDS__INVCODE Invalid type code.
- SDS__INVDIMS Invalid dimensions.
- SDS__NOTSTRUCT Parent is not a structure.
- SDS__EXTERN Parent is external.

SDS_POINTER Get a pointer to the data of a primitive item SDS_POINTER

Description: Return a pointer to the data of a primitive item. Also return the length of the item. If the data item is undefined and the object is internal storage for the data will be created.

If necessary (e.g. if the data originated on a machine with different architecture) the data for the object is converted (in place) from the format stored in the data item to that required for the local machine

Invocation: `CALL SDS_POINTER( ID, PNTR, LENGTH, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object

PNTR = INTEGER (Returned)
Pointer to the data.

LENGTH = INTEGER (Returned)
Actual number of bytes transferred.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID Invalid Identifier.
- SDS__NOTPRIM Not a primitive item.
- SDS__UNDEFINED Data undefined.

SDS_PUT Write data to an object SDS_PUT

Description: Write data into an object. The object may be a primitive item or a structure or structure array.

If the object is a structure or structure array, the data from the the buffer is copied into its primitive components in order of their position in the structure. Alignment adjustments are made as necessary to match the alignment of a C struct equivalent to the SDS structure. In the structure or structure array case the offset parameter is ignored.

If the object is primitive data is transferred starting at the position in the item specified by offset.

If the data was previously undefined memory for the data is allocated at this time.

Invocation: `CALL SDS_PUT( ID, LENGTH, OFFSET, DATA, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object

LENGTH = INTEGER (Given)
Length in bytes of the buffer containing the data.

OFFSET = INTEGER (Given)
Offset into the data object at which to start writing data. The offset is measured in units of the size of each indivdual item in the array - e.g. 4 bytes for an INT or 8 bytes for a DOUBLE. The offset is zero to start at the beginning of the array.

DATA (*) = BYTE (Given)
Buffer containing the data

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID Invalid Identifier.
- SDS__NOMEM Insufficient memory for creation
- SDS__NOTPRIM Not a primitive item.
- SDS__UNDEFINED Data undefined.

SDS_PUTC Write data to a character object SDS_PUTC

Description: Write data into a character object. The object must be a primitive character item.

Data is transferred starting at the position in the item specified by offset.

If the data was previously undefined memory for the data is allocated at this time.

Invocation: `CALL SDS_PUTC( ID, LENGTH, OFFSET, DATA, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object

LENGTH = INTEGER (Given)
Length of the buffer containing the data.

OFFSET = INTEGER (Given)
Offset into the data object at which to start writing data. The offset is zero to start at the beginning of the array.

DATA (*) = CHARACTER (Given)
Buffer containing the data

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID Invalid Identifier.
- SDS__NOMEM Insufficient memory for creation
- SDS__NOTPRIM Not a primitive item.
- SDS__UNDEFINED Data undefined.
- SDS__TYPE Object has incorrect type.

SDS_PUTD Write data to a double precision object SDS_PUTD

Description: Write data into a double precision object. The object must be a primitive double precision item.

Data is transferred starting at the position in the item specified by offset.

If the data was previously undefined memory for the data is allocated at this time.

Invocation: `CALL SDS_PUTD( ID, LENGTH, OFFSET, DATA, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object

LENGTH = INTEGER (Given)
Length of the buffer containing the data.

OFFSET = INTEGER (Given)
Offset into the data object at which to start writing data. The offset is zero to start at the beginning of the array.

DATA (∗) = DOUBLE (Given)
Buffer containing the data

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID Invalid Identifier.
- SDS__NOMEM Insufficient memory for creation
- SDS__NOTPRIM Not a primitive item.
- SDS__UNDEFINED Data undefined.
- SDS__TYPE Object has incorrect type.

SDS_PUTI Write data to an integer object SDS_PUTI

Description: Write data into an integer object. The object must be a primitive integer item.

Data is transferred starting at the position in the item specified by offset.

If the data was previously undefined memory for the data is allocated at this time.

Invocation: `CALL SDS_PUTI( ID, LENGTH, OFFSET, DATA, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object

LENGTH = INTEGER (Given)
Length of the buffer containing the data.

OFFSET = INTEGER (Given)
Offset into the data object at which to start writing data. The offset is zero to start at the beginning of the array.

DATA (∗) = INTEGER (Given)
Buffer containing the data

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID Invalid Identifier.
- SDS__NOMEM Insufficient memory for creation
- SDS__NOTPRIM Not a primitive item.
- SDS__UNDEFINED Data undefined.
- SDS__TYPE Object has incorrect type.

SDS_PUTL Write data to a logical object SDS_PUTL

Description: Write data into a logical object. The object must be a primitive integer item (The SDS kernel does not have a logical type so uses integer items to represent logical values).

Data is transferred starting at the position in the item specified by offset.

If the data was previously undefined memory for the data is allocated at this time.

Invocation: `CALL SDS_PUTL( ID, LENGTH, OFFSET, DATA, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object

LENGTH = INTEGER (Given)
Length of the buffer containing the data.

OFFSET = INTEGER (Given)
Offset into the data object at which to start writing data. The offset is zero to start at the beginning of the array.

DATA (*) = LOGICAL (Given)
Buffer containing the data

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID Invalid Identifier.
- SDS__NOMEM Insufficient memory for creation
- SDS__NOTPRIM Not a primitive item.
- SDS__UNDEFINED Data undefined.
- SDS__TYPE Object has incorrect type.

SDS_PUTR Write data to a real object SDS_PUTR

Description: Write data into a real object. The object must be a primitive real item.

Data is transferred starting at the position in the item specified by offset.

If the data was previously undefined memory for the data is allocated at this time.

Invocation: `CALL SDS_PUTR( ID, LENGTH, OFFSET, DATA, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object

LENGTH = INTEGER (Given)
Length of the buffer containing the data.

OFFSET = INTEGER (Given)
Offset into the data object at which to start writing data. The offset is zero to start at the beginning of the array.

DATA (∗) = REAL (Given)
Buffer containing the data

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID Invalid Identifier.
- SDS__NOMEM Insufficient memory for creation
- SDS__NOTPRIM Not a primitive item.
- SDS__UNDEFINED Data undefined.
- SDS__TYPE Object has incorrect type.

SDS_PUT_EXTRA — Write to the extra information field of an object — SDS_PUT_EXTRA

Description: Write a specified number of bytes to the extra information field of an object. A maximum of 128 bytes may be written to an internal object. It is permissible to write to the extra information field of an external object, but the number of bytes written must not exceed the number originally in the object.

Invocation: `CALL SDS_GET_EXTRA( ID, NEXTRA, EXTRA, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object

NEXTRA = INTEGER (Given)
Number of bytes of extra information.

EXTRA = CHAR (Returned)
Buffer to receive the data.

ACTLEN = INTEGER (Returned)
Actual number of bytes transferred.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID Invalid Identifier.
- SDS__EXTRA Too much extra data.

SDS_RENAME — Change the name of an object — SDS_RENAME

Description: Specify a new name for an object.

Invocation: `CALL SDS_RENAME( ID, NAME, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object

NAME = CHAR (Given)
New name for the object. This should have a maximum length of 15 characters.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID Invalid Identifier.
- SDS__LONGNAME The name is too long.

SDS_RESIZE Change the dimensions of an array SDS_RESIZE

Description: Change the number and/or size of the dimensions of an array. This operation can be performed on primitive arrays or structure arrays. Note that SDS_RESIZE does not move the data elements in the storage representing the array, so there is no guarantee that after resizing the array the same data will be found at the same array index positions as before resizing, though this will be the case for simple changes such as a change in the last dimension only.

Invocation: `CALL SDS_RESIZE( ID, NDIMS, DIMS, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object.

NDIMS = INTEGER (Given)
New number of dimensions.

DIMS (NDIMS) = INTEGER (Given)
Array of dimensions.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID Invalid Identifier.
- SDS__NOMEM Insuficient memory.
- SDS__EXTERN Object is external.
- SDS__NOTARR Object is not an array.
- SDS__INVDIMS Dimensions invalid.

SDS_SIZE Find the buffer size needed to export an object SDS_SIZE

Description: Return the size which will be needed for a buffer into which the object can be exported using the SDS_EXPORT routine.

Invocation: `CALL SDS_SIZE( ID, BYTES, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object.

BYTES = INTEGER (Returned)
Size in bytes of required buffer.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID Invalid Identifier.
- SDS__EXTERN Object is external.

SDS_LIST List contents of an SDS object SDS_LIST

Description: A listing of the contents of an SDS object is generated on standard output. The listing consists of the name type, dimensions and value of each object in the structure. The hierarchical structure is indicated by indenting the listing for components at each level.

For array objects only the values of the first few components are listed so that the listing for each item fits on a single line.

Invocation: `CALL SDS_LIST( ID, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object to be listed.

STATUS = INTEGER (Given and Returned)
Modified Status. SDS_LIST calls a large number of SDS routines so will return error status values if an error occurs in any of these routines.

SDS_READ Read an SDS object from a file SDS_READ

Description: Read an SDS object from a file previously written by SdsWrite. An identifier to an external object is returned. If an internal version of the object is required it can be created using SdsCopy.

SDS_READ must allocate a buffer to read the object into. This should be freeed when you are finished using the object by calling SDS_READ_FREE.

Invocation: `CALL SDS_READ( FILENAME, ID, STATUS )`

Arguments:

FILENAME = CHAR (Given)
The name of the file from which the object will be read.

ID = INTEGER (Given)
Identifier of the external object.

STATUS = INTEGER (Given and Returned)
Modified Status.

- SDS__NOTSDS Not a valid SDS object.
- SDS__NOMEM Insufficient memory for output buffer.
- SDS__FOPEN Error opening the file.
- SDS__FREAD Error reading the file.

SDS_READ_FREE — Free a buffer allocated by SDS_READ — SDS_READ_FREE

Description: SDS_READ allocates a block of memory to hold the external object read in. This memory can be released when the object is no longer required by calling SDS_READ_FREE (note that it is not possible to SDS_DELETE an external object).

If SDS_READ_FREE is given an idientifier which was not produced by a call to SDS_READ/SdsRead it will do nothing.

Deleting the buffer does not free the memory associated with the identifier referencing it. This memory can be freed with the SDS_FREE_ID function.

Invocation: `CALL SDS_READ_FREE( ID, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier to the external object.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID The identifier is invalid.

SDS_WRITE — Write an SDS object to a file — SDS_WRITE

Description: Given an identifier to an internal SDS object, write it to a file. The file can be read back using SDS_READ.

Invocation: `CALL SDS_WRITE( ID, FILENAME, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the object.

FILENAME = CHAR (Given)
The name of the file into which the object will be written.

STATUS = INTEGER (Given and Returned)
Modified Status.

- SDS__BADID The identifier is invalid
- SDS__EXTERN The object is external
- SDS__NOMEM Insufficient memory for output buffer
- SDS__FOPEN Error opening the file
- SDS__FWRITE Error writing the file

D.2 ARG subroutines

ARG_NEW Create a new argument structure ARG_NEW

Description: Creates a new structure to hold arguments and return an identifier to it

Invocation: `CALL ARG_NEW( ID, STATUS )`

Arguments:

ID = INTEGER (Returned)
The identifier to the created structure

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__NOMEM - Insufficient memory.

ARG_PUT0C Put a character string item into an argument structure ARG_PUT0C

Description: A character string item is written into a named component within the specified argument structure. The component is created if it does not currently exist.

Invocation: `CALL ARG_PUT0C( ID, NAME, VALUE, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the argument structure.

NAME = CHARACTER (Given)
The name of the component to be written into.

VALUE = CHARACTER (Given)
The string to be written.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID The identifier is invalid.
- SDS__NOMEM Insufficient memory.
- ARG__NOTSTRING Item is not a string.

ARG_PUT0D Put a double precision item into an argument structure ARG_PUT0D

Description: A double precision item is written into a named component within the specified argument structure. The component is created if it does not currently exist.

Invocation: `CALL ARG_PUT0D( ID, NAME, VALUE, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the argument structure.

NAME = CHARACTER (Given)
The name of the component to be written into.

VALUE = DOUBLE (Given)
The number to be written.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID The identifier is invalid.
- SDS__NOMEM Insufficient memory.
- ARG__NOTSCALAR Item is not a scalar.
- ARG__CNVERR Type conversion error.

ARG_PUT0I Put an integer item into an argument structure ARG_PUT0I

Description: An integer item is written into a named component within the specified argument structure. The component is created if it does not currently exist.

Invocation: `CALL ARG_PUT0I( ID, NAME, VALUE, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the argument structure.

NAME = CHARACTER (Given)
The name of the component to be written into.

VALUE = INTEGER (Given)
The number to be written.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID The identifier is invalid.
- SDS__NOMEM Insufficient memory.
- ARG__NOTSCALAR Item is not a scalar.
- ARG__CNVERR Type conversion error.

ARG_PUT0L Put a logical item into an argument structure ARG_PUT0L

Description: A logical item is written into a named component within the specified argument structure. The component is created if it does not currently exist.

Invocation: `CALL ARG_PUT0L( ID, NAME, VALUE, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the argument structure.

NAME = CHARACTER (Given)
The name of the component to be written into.

VALUE = LOGICAL (Given)
The number to be written.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID The identifier is invalid.
- SDS__NOMEM Insufficient memory.
- ARG__NOTSCALAR Item is not a scalar.
- ARG__CNVERR Type conversion error.

ARG_PUT0R Put a real item into an argument structure ARG_PUT0R

Description: A real item is written into a named component within the specified argument structure. The component is created if it does not currently exist.

Invocation: `CALL ARG_PUTOR( ID, NAME, VALUE, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the argument structure.

NAME = CHARACTER (Given)
The name of the component to be written into.

VALUE = REAL (Given)
The number to be written.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID The identifier is invalid.
- SDS__NOMEM Insufficient memory.
- ARG__NOTSCALAR Item is not a scalar.
- ARG__CNVERR Type conversion error.

ARG_GET0C Get a character item from an argument structure ARG_GET0C

Description: A character string item is read from a named component within the specified argument structure.

Invocation: `CALL ARG_GETOC( ID, NAME, VALUE, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the argument structure.

NAME = CHARACTER (Given)
The name of the component to be read.

VALUE = CHARACTER (Returned)
Character string value.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID The identifier is invalid.
- SDS__NOMEM Insufficient memory.
- ARG__NOTSTRING Item is not a string.

ARG_GET0D Get a double precision item from an argument structure ARG_GET0D

Description: A double floating point item is read from a named component within the specified argument structure. The item is converted to double type if necessary.

Invocation: `CALL ARG_GET0D( ID, NAME, VALUE, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the argument structure.

NAME = CHARACTER (Given)
The name of the component to be read.

VALUE = DOUBLE (Returned)
Double precision value.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID The identifier is invalid.
- SDS__NOMEM Insufficient memory.
- ARG__NOTSCALAR Item is not a scalar.
- ARG__CNVERR Type conversion error.

ARG_GET0I Get an integer item from an argument structure ARG_GET0I

Description: An integer item is read from a named component within the specified argument structure. The item is converted to integer type if necessary.

Invocation: `CALL ARG_GET0I( ID, NAME, VALUE, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the argument structure.

NAME = CHARACTER (Given)
The name of the component to be read.

VALUE = INTEGER (Returned)
Integer value.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID The identifier is invalid.
- SDS__NOMEM Insufficient memory.
- ARG__NOTSCALAR Item is not a scalar.
- ARG__CNVERR Type conversion error.

ARG_GET0L Get a logical item from an argument structure ARG_GET0L

Description: A logical item is read from a named component within the specified argument structure. The item is converted to logical type if necessary.

Invocation: `CALL ARG_GET0L( ID, NAME, VALUE, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the argument structure.

NAME = CHARACTER (Given)
The name of the component to be read.

VALUE = LOGICAL (Returned)
Logical value.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID The identifier is invalid.
- SDS__NOMEM Insufficient memory.
- ARG__NOTSCALAR Item is not a scalar.
- ARG__CNVERR Type conversion error.

ARG_GET0R Get a real item from an argument structure ARG_GET0R

Description: A real item is read from a named component within the specified argument structure. The item is converted to real type if necessary.

Invocation: `CALL ARG_GET0R( ID, NAME, VALUE, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the argument structure.

NAME = CHARACTER (Given)
The name of the component to be read.

VALUE = REAL (Returned)
Real value.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID The identifier is invalid.
- SDS__NOMEM Insufficient memory.
- ARG__NOTSCALAR Item is not a scalar.
- ARG__CNVERR Type conversion error.

ARG_GETDC Get a character item with defaulting ARG_GETDC

Description: A character string item is read from a named component within the specified argument structure. If the item is not present in the structure the default value is returned.

Invocation: `CALL ARG_GETDC( ID, NAME, DEF, VALUE, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the argument structure.

NAME = CHARACTER (Given)
The name of the component to be read.

DEF = CHARACTER (Given)
Default value for argument.

VALUE = CHARACTER (Returned)
Character string value.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID The identifier is invalid.
- SDS__NOMEM Insufficient memory.
- ARG__NOTSTRING Item is not a string.

ARG_GETDD Get a double precision item with defaulting ARG_GETDD

Description: A double floating point item is read from a named component within the specified argument structure. The item is converted to double type if necessary. If the item is not present in the structure the default value is returned.

Invocation: `CALL ARG_GETDD( ID, NAME, DEF, VALUE, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the argument structure.

NAME = CHARACTER (Given)
The name of the component to be read.

DEF = DOUBLE (Given)
The default value.

VALUE = DOUBLE (Returned)
Double precision value.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID The identifier is invalid.
- SDS__NOMEM Insufficient memory.
- ARG__NOTSCALAR Item is not a scalar.
- ARG__CNVERR Type conversion error.

ARG_GETDI Get an integer item with defaulting ARG_GETDI

Description: An integer item is read from a named component within the specified argument structure. The item is converted to integer type if necessary. If the item is not present in the structure the default value is returned.

Invocation: `CALL ARG_GETDI( ID, NAME, DEF, VALUE, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the argument structure.

NAME = CHARACTER (Given)
The name of the component to be read.

DEF = INTEGER (Given)
The default value.

VALUE = INTEGER (Returned)
Integer value.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID The identifier is invalid.
- SDS__NOMEM Insufficient memory.
- ARG__NOTSCALAR Item is not a scalar.
- ARG__CNVERR Type conversion error.

ARG_GETDL Get a logical item with defaulting ARG_GETDL

Description: A logical item is read from a named component within the specified argument structure. The item is converted to logical type if necessary. If the item is not present in the structure the default value is returned.

Invocation: `CALL ARG_GETDL( ID, NAME, DEF, VALUE, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the argument structure.

NAME = CHARACTER (Given)
The name of the component to be read.

DEF = LOGICAL (Given)
The default value.

VALUE = LOGICAL (Returned)
Logical value.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID The identifier is invalid.
- SDS__NOMEM Insufficient memory.
- ARG__NOTSCALAR Item is not a scalar.
- ARG__CNVERR Type conversion error.

ARG_GETDR Get a real item with defaulting ARG_GETDR

Description: A real item is read from a named component within the specified argument structure. The item is converted to real type if necessary. If the item is not present in the structure the default value is returned.

Invocation: `CALL ARG_GETDR( ID, NAME, DEF, VALUE, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the argument structure.

NAME = CHARACTER (Given)
The name of the component to be read.

DEF = REAL (Given)
The default value.

VALUE = REAL (Returned)
Real value.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID The identifier is invalid.
- SDS__NOMEM Insufficient memory.
- ARG__NOTSCALAR Item is not a scalar.
- ARG__CNVERR Type conversion error.

ARG_DELET Delete an argument structure ARG_DELET

Description: Delete an argument structure and free its identifier.

Invocation: `CALL ARG_DELET( ID, STATUS )`

Arguments:

ID = INTEGER (Given)
Identifier of the argument structure.

STATUS = INTEGER (Given and Returned)
Modified Status. Possible failure codes are:

- SDS__BADID The identifier is invalid.

E sdsc Command description

E.1 sdsc — Compiles C structure definitions into SDS Calls.

Function: Compiles C structure definitions into SDS Calls.

Synopsis:

```
sdsc [ -nvlswT] [ -Dname ] [ -Dname=def ] [ -Uname ]
     \[ -ffunction ]  [ -ttype ] [ -Nname ]
     \[ -Idirectory ] [ -Pcpp ] [ input-file ] [ output-file]
```

Description: A C structure can be put and retrived from a similar SDS structure using SdsPut and SdsGet. This makes it desirable to be able to automatically generate SDS calls to produce an SDS structure equivalent to the C structure. sdsc does this job.

The input to sdsc is frist run through a C preprocessor. During this the macros 'SDS' and '___SDS___' will be defined (in addition) to any macros defined by default). The result should be a series of C definitions followed by one and only one structure declaration.

sdsc optionally accepts two filenames as arguments. input-file and output-file are, respectively, the input and output files. They default to the standard input and standard output (except under VMS, when you must specifiy the input file (due to problems with the C run time library under the current version of VMS)).

Options:

-n Do not run the input through the C preprocessor.

-v Output the command used to run the C preprocessor to stderr.

-l Treat ambiguous int declarations as 32bit ints.

-s Treat ambiguous int declarations as short ints.

-w Instead of outputing declarations, use the SdsWrite utility function to write the Sds structure to a file . In this case, an output file must be supplied.

-W Output warning messages as well as error messages.

-T Output Tcl code instead of C code. This requires the Sds commands in DRAMA's Dtcl package (DRAMA V1.2 and later only)

-Dname Define name as 1 (one). This is the same as if a -Dname=1 option appeared on the command line, or as if a

```
#define name 1
```

appeared in the source file.

-ffunction Specify a function name. When specified, instead of outputing just the body of the function, then entire function is output, this is the name for the function.

-ttype Specify a type, a declaration of which is to be assumed at the end of reading the input file if the input fie does not declare a variable. It should be a simple type or a typedef name.

-Nname If -t is used, then this name is given to the variable created and hence to the top level structure. If not supplied, then the variable type is used as the variable name.

-Dname=def Define name as if by a #define directive.

-Uname Remove any initial definition of name where name is a symbol predefined by the C preprocessor.

-Idirectory Insert directory into the search path for #include files.

-Pcpp Use cpp as the C preprocessor. By default /usr/lib/cpp is used (utask_Dir:gnu_cpp on a vms machine)

Note that the `-D`, `-U` and `-I` options are passed directly to the C preprocessor. If -n is set, none of the other options have any effect (and C preprocessor statements may call errors) .

See Also: cpp(1), Sds manual.

Support: Tony Farrell, `AAO`

F SDS Data Format

F.1 Overall Structure

An SDS structure consists of an array of longwords (one longword = 4 bytes). Each longword can be considered as having a longword address within the block, with the first longword having address 0. The array is organized into three sections as follows:

1. A header, occupying the first four longwords.
2. The definition part.
3. The data part.

The definition part is always present. The data part can be absent if the structure has no defined primitive data items.

F.2 The Header

The header consists of four longwords as follows:

1. The first longword has the value zero if integer items within the header and definition have big endian byte order (i.e. the most significant byte at the lowest address), and has the hex value FFFFFFFF if the integer items have little endian byte order. Any other value is illegal.

   Note that this flag applies to the integer items (pointers, item counts etc) within the header and definition, not to the data in the structure. Each primitive data item has its own format flag in the definition part of the structure.
2. The second longword is the total length in bytes of the structure (including header plus definition plus data).
3. The third longword contains the current SDS version code. The current value for this is the constant SDS__VERSION in the sds.h include file.
4. The fourth longword is the length in bytes of the header plus definition (but not including the data). It is thus the address of the start of the data section.

F.3 The Definition Part

The definition part consists of a sequence of blocks, each corresponding to an object in the data structure. The blocks are laid out according to the following rules:

1. The block describing the top level object must be located immediately following the definition, i.e. starting at longword address 4.
2. The blocks describing the components of a structure or structure array must immediately follow the block describing the parent structure or structure array. They should be in order of position in the structure.

The following example illustrates the sequence of blocks:

```
1 2A 3A 3B 2B 3C 3D 3E 2C
```

In this case the top level object (1) is a structure with three components (2A, 2B, 2C). Component 2A is a structure with two components (3A, 3B) and 2B is a structure with three components (3C, 3D, 3E). 2C is a primitive.

F.3.1 Blocks

The usage of the first five longwords is common to all three block types:

Longword zero is divided up as follows:

- Byte 0 contains the type code for the object with possible values as follows:

Type	Value
STRUCT	0
CHAR	1
BYTE	2
UBYTE	3
SHORT	4
USHORT	5
INT	6
UINT	7
FLOAT	8
DOUBLE	9
SARRAY	10
INT64	11
UINT64	12

  Where SARRAY is the code for a structure array.

- Byte 1 contains the format code for the object.

  Formats for integer items are as follows:

Format	Value
Big endian	0
Little endian	1

  Formats for floating point items are as follows:

Format	Value
IEEE	0
VAX	1
IEEE reversed	2
VAXG	3

  Where IEEE refers to IEEE-754 32 bit format for float and 64-bit format for double type. VAX refers to VAX F-floating format for float, and VAX D-floating format for doubles. VAXG is VAX G format for doubles. IEEE reversed is the IEEE format as implemented on machines with little endian byte order.

- Bytes 2 and 3 constitute a 16 bit integer which is the number of items for a structure, or the number of dimensions for a primitive or structure array.

Longwords 1 to 4 contain the name of the item as a null terminated C string. Since the null must be included the maximimum length of the name is 15 characters.

F.3.2 Structure Blocks

Starting at longword 5 is an array of pointers (i.e. longword addresses relative to the start of the structure) of the components of the structure.

Following this (i.e. starting at longword 5+n where n is the number of components) is the extra information field. This consists of a 16 bit integer specifying the number (nextra) of extra bytes, followed by nextra bytes of information.

The total length (in bytes) of a structure block is:

$$22 + 4 * nitems + nextra$$

though it will always be padded to a whole number of longwords so that the next block begins on a longword boundary.

F.3.3 Primitive Blocks

Longword 5 of a primitive block is the longword address of the data (i.e. a pointer into the data part. If the data is undefined this longword will be zero.

Starting at longword 6 is an array of longwords containing the dimensions of the array.

Following this (i.e. starting at longword 6+n where n is the number of dimensions) is the extra information field. This consists of a 16 bit integer specifying the number (nextra) of extra bytes, followed by nextra bytes of information.

The total length (in bytes) of a primitive block is:

$$26 + 4 * ndims + nextra$$

though it will always be padded to a whole number of longwords so that the next block begins on a longword boundary.

F.3.4 Structure Array Blocks

Starting at longword 5 is an array of longwords containing the dimensions of the array.

Following this (i.e. starting at longword 5+ndims) is an array of pointers (longword addresses) to the elements of the structure array, each of which has its own structure block. These pointers are the longword addresses of the start of the corresponding structure block.

Following this (i.e. starting at longword 5+ndims+n where n is the number of elements) is the extra information field. This consists of a 16 bit integer specifying the number (nextra) of extra bytes, followed by nextra bytes of information.

The total length (in bytes) of a structure array block is:

$$22 + 4 * ndims + 4 * nelements + nextra$$

though it will always be padded to a whole number of longwords so that the next block begins on a longword boundary.

F.4 The Data Part

The Data Part contains the data for each primitive object in the same order in which the definition blocks are stored. Each item starts on a longword boundary and its size is the size of the primitive type multiplied by the number of array elements. Note that undefined objects do not have any associated data.

Each object of type double has an additional longword allocated to it, which is used as padding to ensure that the data for the object begins on an eight byte boundary. This padding longword is positioned either before or after the data for the object to give the correct alignment. Thus the size of a double item is four bytes larger than actually required to store the data.