M.O.P.S. — FORTRAN 77 USER GUIDE

19 SEP. 1989

Werner Herr and Rödiger Schmidt

SPS Division, CERN, Geneva, Switzerland

VERSION 3.00 (March 1989)

(Includes latest version: 3.04)
CONTENTS

1. INTRODUCTION ................................................. 1

2. THE BASIC CONCEPT .............................................. 1

3. MOPS FOR BEGINNERS ............................................ 3
   3.1 Initialisation of the data structure ....................... 3
   3.2 Booking an object in the data structure ................. 4
   3.3 Writing data into the data structure .................... 5
   3.4 Accessing data in the data structure .................... 6

4. THE SERVICE PROGRAMS ......................................... 8
   4.1 Sequential input/output of data structures ............. 8
   4.2 Object manipulation .................................. 10
      4.2.1 Dropping objects and garbage collection ........ 10
      4.2.2 Changing the size of an object .................. 11
   4.3 Testing and status report ............................. 11
      4.3.1 Finding an object ................................ 12
      4.3.2 General information on the data structure ....... 13
   4.4 Plotting and histogramming ............................. 13
   4.5 Tools for debugging ................................... 15
      4.5.1 Error output .................................... 15
      4.5.2 MOPS return codes ................................ 15
      4.5.3 Hexadecimal dump of a data structure .......... 16
      4.5.4 Validity check .................................. 16
      4.5.5 Copying a data structure ......................... 16
      4.5.6 Comparing two data structures .................... 16
   4.6 Building relational data structures ..................... 17
   4.7 Network transfer ....................................... 18

5. RUNNING INSTRUCTIONS ....................................... 19
   5.1 IBM VM/CMS ............................................ 19
   5.2 NORD 500 SINTRAN ..................................... 19
   5.3 APOLLO workstations .................................. 20
   5.4 VAX VMS ............................................... 20
   5.5 Technical details ..................................... 20

6. APPENDIX A .................................................. 22

7. APPENDIX B .................................................. 24

8. REFERENCES .................................................. 27

Acknowledgments ............................................... 28

INDEX OF MOPS SUBROUTINES: .................................... 29

GENERAL INDEX .................................................. 30
1. INTRODUCTION

The MOPS system\(^1\) is a set of service routines which give the users the possibility to organise and manage their data in a structured fashion. Since their introduction in 1986 [1, 2] the MOPS data structure conventions and support programs have been used in a large number of programs and applications and have been installed on many different computer systems at CERN. They have been implemented in FORTRAN 77 and "C" but the underlying data structures\(^2\) and concepts for both, the "C" version [1] and the FORTRAN 77 version [2], are identical and allow the exchange of data between programs written in these languages.

For both languages the MOPS data structure management system has been developed further and new facilities have been added and existing service programs have been improved and extended. This report will try to serve as an introduction for novice users as well as a reference manual for experienced users and it supersedes [2] as a user guide.

Since a description has been given in previous reports [1, 2, 3] we shall only briefly review the basic concepts in the second chapter and concentrate on a simplified guide for "first time users" in the third chapter and a detailed description of all available service programs in the following chapters. We shall try to include examples wherever possible to clarify the use of these routines.

2. THE BASIC CONCEPT

The physical location of the data structure we shall call the "dynamic store" which is a contiguous part of the memory. This is obtained from the system by declaring one or more variable arrays which can be of any type but for convenience they should be declared as type INTEGER+4. Problems can occur with a store declared as a CHARACTER array on computers with VAX-like string descriptors, such as NORD 500 or VAX VMS systems. A MOPS data structure is mapped onto this dynamic store (array). Our implementation allows the simultaneous use of several such data structures stored in different dynamic stores, i.e. different arrays.

The basic unit of the data structure is an entity we call an "object" which contains a number of data elements grouped together according to logical affinity (e.g. \(\beta\) functions) and stored in contiguous storage words in the dynamic store. Such an "object" is the basic unit which can be manipulated by the users. Every object can contain any number of data items ("elements"), for example floating point variables or integer numbers. In particular, another MOPS data structure can be stored as one object in a data structure.

The data structure is arranged as a partitioned data set with a "directory" at the beginning followed by one or more user objects which are described in the directory. A simple example for such a data structure is illustrated in Fig.1.

It shows a data structure with two user objects, U01 and U02, and the directory in front of it (DIR). The directory contains all information necessary to describe the data structure and to access the user objects. In the example in Fig.1 the directory has three entries, one for each user object and one for the directory itself. The service routines and tools provided are used to manipulate and access the different data objects and the directory. The directory should be considered as a "system object" in contrast to the "user objects". For every object its entry in the directory has to be provided and filled partly by the user and partly by the MOPS system. This entry contains the following information:

---

\(^1\) Multiple Object Partitioned Structure

\(^2\) In our understanding a data structure is a collection of data, a description of the data and the logical relations between the data items.
- Offset to the user object in bytes (calculated and filled in by the system).
- Keyword or object name (assigned by the user).
- Element type (valid types are: INTEGER*4, INTEGER*2, REAL*4, REAL*8, COMPLEX*8, COMPLEX*16, CHARACTER, MOPS and UNDEFINED).
- Element code (an integer number, derived from element type by the system).
- Number of elements for this object (e.g. number of integer variables etc., defined by the user).
- Size of one element in bytes (e.g. 4 for INTEGER*4 or REAL*4, supplied by the user).
- Timestamps (filled in by the system).

Fig.1: Schematic layout of a MOPS data structure.

These parameters are arranged as a structure and for every user object one such structure (entry) is created in the directory part of the data structure. In Fig.1 the directory entries are expanded for all objects to illustrate this scheme.
Different user objects can hold data of different type and the description of these objects in the directory allows simple manipulations for transfer across networks. Apart from these features the concept of accessing data via a keyword rather than through variable or array names was one of the basic aims to simplify its use and the portability between different high-level languages.

3. MOPS FOR BEGINNERS

In this chapter we shall try to show how to work with MOPS by introducing the basic routines which should be sufficient to write simple programs using the MOPS package and we will try to smooth the way by means of simple and realistic examples.

The user is supposed to be somewhat familiar with the basic ideas but no knowledge of the technical details is required. Important concepts will be repeated as we go along if necessary.

To create and use a MOPS data structure the following steps have to be performed:

- Definition of the dynamic store
- Initialisation of the data structure
- Making a directory entry for an object (booking)
- Writing data into an object
- Accessing data in an object

3.1 Initialisation of the data structure

The very first step is to reserve space for the data structure by defining the dynamic storage. This is done by a declaration of a FORTRAN array which preferably should be of the type INTEGER*4, e.g.:

```fortran
INTEGER*4 Q(100000)
```

This will reserve 400 kbytes for the dynamic store to be used by the data structure. The user has to make sure that this array is big enough to hold the complete data structure.

This declaration will be assumed for all our examples.

The next step is the initialisation of the data structure which is performed with a routine:

```
SUBROUTINE SDINI (NAME, NOBJ, Q)
```

```
CHARACTER*80 NAME  /* Name of data structure */
INTEGER*4 NOBJ  /* Number of user objects */
INTEGER*4 Q  /* Address of dynamic store */
```
where NAME is the name of the data structure, NOBJ the maximum number of user objects the data structure should hold and Q is the address of the dynamic store (i.e. in FORTRAN the name of the array). For example:

\[
\text{CALL SDINI('TWISS', 25, Q)}
\]

would reserve space for 25 user objects in a data structure with the name 'TWISS' and the data structure is mapped into the dynamic store addressed by Q. If the NAME of the data structure is given as a variable and not as a immediate variable (character constant) as in the example above, it has to be declared as a CHARACTER\text{*n} variable where n can be anything between 1 and 80. After the successful execution of this call space has been reserved for the directory itself and the dynamic store is now recognised as a valid MOPS data structure by all service programs.

3.2 Booking an object in the data structure

Before any user data can be put into an object in the data structure an entry into the directory has to be made for this user object\textsuperscript{3} in order to:

- Reserve enough space to hold the data.
- Calculate the offset to the object.
- Assign a name (keyword) to the object.
- Define the type of the elements in this object.
- Associate timestamps with the object.

Such an entry is made by a call to:

```
SUBROUTINE SDBOOK (NELEMS, ELEMSZ, NAME, TYPE, Q)
```

| INTEGER*4 | NELEMS | /* Number of elements */ |
| INTEGER*4 | ELEMSZ | /* Size of one element in bytes */ |
| CHARACTER*80 | NAME | /* Name of the data object */ |
| CHARACTER*40 | TYPE | /* Type of data */ |
| INTEGER*4 | Q | /* Address of dynamic store */ |

The TYPE of elements available in this version are INTEGER*2, INTEGER*4, REAL*4, REAL*8, DOUBLE, COMPLEX*8, COMPLEX*16, CHARACTER and UNDEFINED. The element type MOPS must be used to store a complete MOPS data structure as an object. The variables NAME and TYPE can be replaced by character constants in the calling sequence but their length must be smaller than 80 or 40 characters. This is possible for all calls to MOPS routines which have character strings as parameters.

\textsuperscript{3} The entry for the directory is made during the initialisation.
Imagine we wish to store all information about the horizontal β functions in a single object, we can use

\[
\text{CALL SDBOOK(144, 4, 'BETAX', 'REAL*4', Q)}
\]

to reserve space for 144 floating point variables in the data structure for an object with the name (keyword) 'BETAX'.

To store text in the data structure one would use a call like:

\[
\text{CALL SDBOOK(100, 80, 'COMMENTS', 'CHARACTER', Q)}
\]

The offset (pointer) to the user object is calculated by the system and stored in the directory as well as the date and time.

A call to SDBOOK only reserves space but does not initialise the contents of the object. An additional call is provided if the user object has to be initialised to zero:

```
SUBROUTINE SDZERO (NELEMS, ELEMSZ, NAME, TYPE, Q)
```

- INTEGER*4 NELEMS /* Number of elements */
- INTEGER*4 ELEMSZ /* Size of one element in bytes */
- CHARACTER*80 NAME /* Name of the data object */
- CHARACTER*40 TYPE /* Type of data */
- INTEGER*4 Q /* Address of dynamic store */

After an object has been booked it can be manipulated using the keywords (i.e. 'COMMENTS' or 'BETAX'). The simplest manipulation would be to enter data into the structure or to access data from the structure.

### 3.3 Writing data into the data structure

After space has been reserved for a user object, data can be written into the structure. If the data is stored locally in an array INN(n) or if it starts at the address INN the data is copied into the structure by a call like:

```
SUBROUTINE SDWRIT(NAME, NEL, INN, Q)
```

- CHARACTER*80 NAME /* Name (keyword) of already booked object */
- INTEGER*4 NEL /* Number of elements to be copied */
- any type .. INN /* Local array holding the data */
- INTEGER*4 Q /* Address of dynamic store */

where NAME is the object name as registered with a call to SDBOOK, NEL is the number of elements to be copied, INN is the array name or start address of the data and Q is the address (name) of the array holding the data structure. To fill our object 'BETAX' with data, the call:

\[
\text{CALL SDWRIT('BETAX', 144, MEBETA, Q)}
\]
would be used, where ‘MEBETA’ is a local array in the program. The number of elements NEL to be stored must be smaller or equal to the number booked with SDBOOK() or SDZERO(), otherwise an error message will be issued and the request is ignored.

It is possible to fill an object starting at a location other than the beginning of the object with

```
SUBROUTINE SDPOKE(NAME, SEL, NEL, INN, Q)

  CHARACTER*80  NAME          /* Name (keyword) of already booked object */
  INTEGER*4    SEL           /* Number of element where copy should start */
  INTEGER*4    NEL           /* Number of elements to be copied */
  any type     INN           /* Local array holding the data */
  INTEGER*4    Q             /* Address of dynamic store */
```

where SEL is now the location of the first element in the object which will be written. If SEL is set to 1, SDPOKE() and SDWRIT() are equivalent.

Calls to SDBOOK() and SDWRIT() can be performed at any time within the program although subsequent calls to SDBOOK() with identical keywords will result in an error. Subsequent calls to SDWRIT() for the same object will overwrite the data in the object. In contrast to ref. [1] the complete directory structure need not to be defined before the first object can be filled with data. The structure can therefore be defined at execution time of the program.

### 3.4 Accessing data in the data structure

We have provided two different methods to access data in a data structure. The call:

```
SUBROUTINE SDPTR(NAME, POINT, Q)

  CHARACTER*80  NAME          /* Name of user object */
  INTEGER*4    POINT          /* Pointer into data structure */
  INTEGER*4    Q              /* Address of dynamic store */
```

returns an integer pointer POINT to the user object with the keyword NAME (pointing into the array Q). The first data element of the object NAME would then be Q (POINT). We could get access to our $\beta$ functions will

```
CALL SDPTR('BETAX', LP, Q)
```

and Q(LP) ... Q(LP + 143) are the values of our $\beta$ function.

If the data elements are not 4 bytes long, the pointer is normalised to the element size. E.g. if the stored data is of type REAL*8, the returned pointer has the correct value for an array defined as REAL*8 and equivalenced with the dynamic store as:

```
REAL*8     QR8(50000)
EQUIVALENCE (Q(1), QR8(1))
```
and POINT would point to the beginning of the data object in QR8, i.e. QR8(POINT). The access to stored character strings is more difficult and another access method should be used (SDREAC).

It should be pointed out here that this call can also be used to enter data into an object since the pointer into the data structure is provided and no data locking is foreseen. No type check and no size check is done whereas SDWRIT() would not allow the user to enter data of the wrong type or too many data elements.

If a user wants to have a copy of the object into a local array a call to

```
SUBROUTINE SDREAC(NAME, POINT, OUT, Q)

  CHARACTER*80  NAME  /* Name of user object */
  INTEGER*4    POINT  /* Pointer into data structure */
  any type     OUT    /* Local array to receive data */
  INTEGER*4    Q      /* Address of dynamic store */
```

can be used, where the entire user object NAME is copied into the local array OUT. It is the users responsibility to make sure that the array OUT is dimensioned large enough to hold the complete object.

It should be mentioned that the array OUT will always contain the correct data independent of the data type used for the array Q. A problem can be the access of character strings stored in the data structure since the character string descriptors are not standardised in FORTRAN 77 and strings are stored in a machine independent way in the MOPS data structure. Therefore the access with SDPTR() can lead to unpredictable results but SDREAC() would always give the correct data. A call:

```
CALL SDREAC('BETAX', POINT, OUT, Q)
```

would be used to copy the β functions stored in the data structure into a local array OUT.

To copy only part of an entire object the subroutine SDSNAP is provided:

```
SUBROUTINE SDSNAP(NAME, POINT, SEL, NEL, OUT, Q)

  CHARACTER*80  NAME  /* Name of user object */
  INTEGER*4    POINT  /* Pointer into data structure */
  INTEGER*4    SEL    /* First element to be copied from object */
  INTEGER*4    NEL    /* Number of elements to be copied from object */
  any type     OUT    /* Local array to receive data */
  INTEGER*4    Q      /* Address of dynamic store */
```

All elements between element number SEL and SEL + NEL are copied into the array OUT. It is the users responsibility to make sure that the array OUT is dimensioned large enough to hold the complete object. A call:

```
CALL SDSNAP('BETAX', POINT, 51, 50, OUT, Q)
```

* This feature is restricted to certain common types and computers.
would only copy the elements 51 ... 100 from object BETAX into OUT.

If the object with the name NAME has been found, a copy of the directory entry is made into a COMMON /DIREC/ (see 5.5) and the attributes of the data object are available. For example, the number of elements are found in the variable NELEMS, the element size is store in ELEMSZ (see 5.5). This common block is available if the MOPS include sequence is included into the calling subroutine.

We are now prepared to write a small FORTRAN 77 program using the MOPS data structure and a simple example for such a program is shown in appendix A to illustrate the use of the MOPS system calls we have learned on our way so far.

4. THE SERVICE PROGRAMS

So far all basic concepts and tools concerning the data management have been presented, this is sufficient to start planning and writing programs. Further extra service routines are described in this chapter. Some of the tools the users might find very useful to structure their programs and to increase readability and flexibility of their code.

4.1 Sequential input/output of data structures

An important feature is the possibility to copy the data structure as an entity to an external medium (disk, floppy disk or tape) or to load it back into memory. The internal structure and the integrity of the data structure must be maintained during this operation. A standard format is used to ensure the proper representation of data and the writer of programs should not have to worry about the hardware representation of variables. Currently the file access is only sequential, but a random access procedure is planned for the next version of this package.

To copy the complete data structure held in the dynamic store Q to a file with the name FNAME the routine SDFIL() is used:

```fortran
SUBROUTINE SDFIL(FNAME, Q)
  CHARACTER*80   FNAME  /\ Name of file */
  INTEGER*4      Q      /\ Address of dynamic store */
END
```

Our data structure would be stored on a file 'sps.optics' with

```fortran
CALL SDFIL('sps.optics', Q)
```

A subsequent call (possibly in another program) to

```fortran
SUBROUTINE SDUFIL(FNAME, Q)
  CHARACTER*80   FNAME  /\ Name of file */
  INTEGER*4      Q      /\ Address of dynamic store */
END
```
would recover the data structure stored in the file FNAM1; and read it as an entity into the array Q.

In case it should be required to refer to the file with a logical unit identifier (this can be convenient on
the IBM), these two calls can be replaced by calls to:

\[
\begin{align*}
\text{SUBROUTINE SDSTO(LUN, Q)} & \quad \text{and} \\
\text{SUBROUTINE SDUSTO(LUN, Q)}
\end{align*}
\]

\[
\begin{align*}
\text{INTEGER}^*4 & \quad \text{LUN} & \quad /\!* \text{logical unit identifier }*/ \\
\text{INTEGER}^*4 & \quad \text{Q} & \quad /\!* \text{Address of dynamic store }*/
\end{align*}
\]

A FORTRAN open statement for the unit LUN must precede a call to SDSTO() or SDUSTO(). All four I/O routines described would store the data structure as a binary data set to increase efficiency.

Should it be necessary to store the data structure as an ASCII file on an external medium and to re-
trieve it, two calls are available

\[
\begin{align*}
\text{SUBROUTINE QTOASC(Q, LUN)} \\
\text{SUBROUTINE ASCTOQ(Q, LUN)}
\end{align*}
\]

\[
\begin{align*}
\text{INTEGER}^*4 & \quad \text{LUN} & \quad /\!* \text{logical unit identifier }*/ \\
\text{INTEGER}^*4 & \quad \text{Q} & \quad /\!* \text{Address of dynamic store }*/
\end{align*}
\]

Since the data type is known by the system, real data will be printed as real and integer data as integer
numbers which allows an easy control of the contents\(^*\) of the data structure. In front of the data the
directory structure is printed in a standard format by QTOASC() and this allows a subsequent call to
ASCTOQ() to recover the entire data structure into the dynamic store. A FORTRAN open statement
for the unit LUN must precede a call to QTOASC() or ASCTOQ().

The routine

\[
\text{SUBROUTINE QTOINF(Q, Z, NWORDS)}
\]

\[
\begin{align*}
\text{INTEGER}^*4 & \quad \text{Q} & \quad /\!* \text{Address of dynamic store }*/ \\
\text{CHARACTER}^*80 & \quad \text{Z} & \quad /\!* \text{String array to receive data structure }*/ \\
\text{INTEGER}^*4 & \quad \text{NWORDS} & \quad /\!* \text{Number of records copied }*/
\end{align*}
\]

works as QTOASC() but writes the output into an internal file specified as a string array of record
length 80 ( Z ). The number of records, i.e. number of lines, is returned in the variable NWORDS.

This could be useful to store internally the data as an ASCII file, i.e. in computer independent repre-
sentation, if the data structure should be transferred across a network with primitive data conversion
capabilities.

\(^*\) Nothing written if data type is UNDEFINED.
On UNIX SYSTEM V installations an additional mode to store the data structure is possible. It can be stored in a named shared memory segment SDNAME [1] with a call to:

```
SUBROUTINE SDPUT(SDNAME, Q)
```

```
CHARACTER*80 SDNAME: /* Name of shared memory segment */
INTEGER*4 Q: /* Address of dynamic store */
```

and retrieved with:

```
SUBROUTINE SDGET(SDNAME, Q)
```

```
CHARACTER*80 SDNAME: /* Name of shared memory segment */
INTEGER*4 Q: /* Address of dynamic store */
```

To store the data structure on a shared memory segment (and make it available to "C" programs)

```
CALL SDPUT('sps.twiss', Q)
```

could be used. These calls are not available on none UNIX SYSTEM V computers. The user should consult ref. [3] before using these utilities.

### 4.2 Object manipulation

In this section we shall describe features which did not exist in the older version, namely the possibility to manipulate particular objects at runtime. This manipulation includes the change of the size of an object, dropping an object and a garbage collection facility.

#### 4.2.1 Dropping objects and garbage collection

As objects are created the occupied space increases and the free space in the data structure decreases. Any object and the associated space in the dynamic store is released by dropping an object: if the data contained in an object is not needed any longer, the object can be marked as "dropped" by a call to

```
SUBROUTINE SDDROP(NAME, Q)
```

```
CHARACTER*80 NAME: /* Name of objects to be dropped */
INTEGER*4 Q: /* Address of dynamic store */
```

A call to SDDROP() logically removes the object from the data structure but dropped objects stay in memory until a garbage collection is triggered either by the system or by the user with a call to

```
SUBROUTINE SDGARB(Q)
```

```
INTEGER*4 Q: /* Address of dynamic store */
```
which squeezes out dropped objects and shifts live ones which then form a contiguous area again. This moving of objects requires updates of all offsets and links and this is done automatically by the garbage collection, i.e. a bias is added to all offsets and link addresses. This involves extensive copying and could create some time overhead. The MOPS memory manager triggers a garbage collection automatically whenever a request for space cannot be satisfied and it should normally not be called by the user except in special cases. To erase an entire data structure a call to SDINIT() is far more efficient.

4.2.2 Changing the size of an object

A facility which has been introduced into this new version is the possibility to change the size of an object whenever this is required at runtime. The size of an object can be decreased with

```
SUBROUTINE SDCUT( NAME, NEWLEN, Q)

    CHARACTER*80 NAME /* Name of object to be changed */
    INTEGER*4 NEWLEN /* New number of objects for this element */
    INTEGER*4 Q /* Address of dynamic store */
```

where NEWLEN is the new number of elements in this object. This number must be smaller than the original size, otherwise an error message will be issued and no action is performed.

To increase the size of an object one would use

```
SUBROUTINE SDAUGM( NAME, NEWLEN, Q)

    CHARACTER*80 NAME /* Name of object to be changed */
    INTEGER*4 NEWLEN /* New number of objects for this element */
    INTEGER*4 Q /* Address of dynamic store */
```

In this case NEWLEN must be larger than the original size, otherwise an error message is printed and the request is ignored.

Both actions are performed in situ and all offsets and links are updated as necessary to accommodate the new size of the object NAME inside the data structure when these routines are executed. The size of the directory cannot be changed with these calls. Both routines perform heavy data copying and should not be used in time critical parts of the program.

4.3 Testing and status report

The following routines can help to structure the user programs and to allow the user to access attributes of objects or of the entire data structure. Their use is recommended since it increases the readability and flexibility.
3.1 Finding an object

The subroutine

```
SUBROUTINE SDFIND(NAME, I, Q)
  CHARACTER*80  NAME  /* Name of the object to be tested */
  INTEGER*4    I     /* Sequence number of NAME in data structure */
  INTEGER*4    Q     /* Address of dynamic store */
```

is used to test whether an object NAME exists in the data structure. If it exists, the sequence number, i.e. the relative position of the object in the data structure is returned and a copy of the directory entry is made into the common block COMMON /DIREC/ (see chapter 5.5). After a call:

```
CALL SDFIND('BETAX', I, Q)
```

the number of elements is available as NELEMS and would be 144 in our example.

If the object NAME does not exist in the data structure a value of $I = -1$ is returned.

If the sequence number is known or if all object are tested in sequence the subroutine

```
SUBROUTINE SDGREP(I, Q)
  INTEGER*4    I     /* Sequence number of NAME in data structure */
  INTEGER*4    Q     /* Address of dynamic store */
```

can be used. A copy of the $I^{th}$ directory entry is made into the COMMON /DIREC/.

The knowledge of the sequence number allows an alternative access method. The routines

```
SUBROUTINE SDREAX(I, POINT, Q)
SUBROUTINE SDREAY(I, POINT, OUT, Q)
  INTEGER*4    I     /* Sequence number of NAME in data structure */
  INTEGER*4    POINT /* Pointer into data structure */
  any type     OUT   /* Local array to receive data */
  INTEGER*4    Q     /* Address of dynamic store */
```

are functionally equivalent to SDPTR() and SDREA() except that the sequence number $I$ is used instead of the object name.
4.3.2 General information on the data structure

To total length of the data structure in Q is returned by the function

```FUNCTION SDLEN(Q)```

INTEGER*4 Q   /\* Address of dynamic store */

The directory entries are used for this calculation and a check on the validity of the offsets is done at the same time.

To obtain the status of the entire data structure one would use

```SUBROUTINE SDSTAT(Q)```

INTEGER*4 Q   /\* Address of dynamic store */

which fills a common block COMMON /SDSTAT/ which is described in chapter 5.5.

4.4 Plotting and histogramming

When the data structure is initialised two objects are already prepared: the directory and the so-called plot object which is normally not visible or directly accessible to the user. The plot object is used to store plotting information if an object should be plotted either by the SPS dataviewer [4] or by another graphics package. To initialise a plot for a particular object, an entry into this plot object has to be made using one of the system calls available. An entry for a plot or histogram in the plot object is made with a call to either SDPLOT() or SDHIST(). Since the calling sequence is the same for both routines, we shall only describe the use of SDPLOT():

```SUBROUTINE SDPLOT( NAME, UNIT, YLOW, YHIGH, SCALE, Q)```

CHARACTER*80 NAME   /\* Object to be plotted */
CHARACTER*80 UNIT    /\* Unit used for x axis */
REAL*4 YLOW    /\* Lower limit for x axis */
REAL*4 YHIGH    /\* Upper limit for x axis */
REAL*4 SCALE   /\* Scale for x axis */
INTEGER*4 Q     /\* Address of dynamic store */

The so-called plot index (see 5.5) is automatically updated by such a call and the plot object is dynamically expanded if necessary. For example

```CALL SDPLOT('BETAX', 'beta function', 0., 500., 1.0, Q)```

would produce an entry in the plot object to prepare a plot of the \( \beta \) function stored in the object \textit{BETAX} with lower and upper limits of 0.0 and +500 metres.

A routine SDPLTC() is used to produce two dimensional plots:
SUBROUTINE SDPLTC( XNAME, XUNIT, XLOW, XIHIGH, XSCALE, &
YNAME, YUNIT, YLOW, YIHIGH, YSCALE, Q)

CHARACTER*80 XNAME        /* Object to be plotted */
CHARACTER*80 XUNIT        /* Unit used for x axis */
REAL*4 XLOW            /* Lower limit for x axis */
REAL*4 XIHIGH           /* Upper limit for x axis */
REAL*4 XSCALE          /* Scale for x axis */
CHARACTER*80 YNAME        /* Object to be plotted against XNAME */
CHARACTER*80 YUNIT        /* Unit used for y axis */
REAL*4 YLOW            /* Lower limit for y axis */
REAL*4 YIHIGH           /* Upper limit for y axis */
REAL*4 YSCALE          /* Scale for y axis */
INTEGER*4 Q            /* Address of dynamic store */

To obtain the normal HBOOK printout for these histograms, a call to

SUBROUTINE SDDRAW(Q)

INTEGER*4 Q        /* Address of dynamic store */

is used. The HBOOK output is directed to the standard output device.

To plot the output on a graphics terminal one would call

SUBROUTINE SDHPLO(NAME, WSTYP, Q)

CHARACTER*80 NAME       /* Name of object to be plotted */
INTEGER*4 WSTYP         /* Workstation type, e.g. 10004 = APOLLO */
INTEGER*4 Q             /* Address of dynamic store */

The type of the workstation has to be given to enable the correct initialisation of GKS calls, e.g. 10004 for APOLLO DN3000/DN4000 or 7878 for P1:RICOM graphics. The plot is held on the terminal until <CR> is pressed.

To print a plot on a laser printer, a GKS metafile has to be produced and this is done with

SUBROUTINE SDMETA(NAME, LUN, Q)

CHARACTER*80 NAME       /* Name of object to be plotted */
INTEGER*4 LUN           /* Logical unit for GKS metafile */
INTEGER*4 Q             /* Address of dynamic store */

This metafile can be printed on a APA6670 or 3812 laser printer. A FORTRAN open statement for the unit LUN must precede a call to SDMETA(). GKS metafiles produced on a VAX or APOLLO can be transferred via FTP to IBM VM/CMS if no laser printer is connected to the workstation.
A PAW direct access file is created with

```
SUBROUTINE SDPAW(Q)

INTEGER 4  Q  / Address of dynamic store */
```

and it can be manipulated with standard PAW commands.

### 4.5 Tools for debugging

#### 4.5.1 Error output

All MOPS error messages are normally printed to the standard output device, but the user can redefine the error output unit with

```
SUBROUTINE SDERR(LUN)

INTEGER 4  LUN  / Logical unit number for error output */
```

and all errors are printed to the device with the logical unit number LUN. A FORTRAN open statement for the unit LUN must precede a call to SDERR() if LUN is not the standard output device.

Usually all MOPS errors lead to the termination of the program if no special precautions are taken. The termination level can be changed by setting a variable NSTOPT to a value between 0 and 2. The default value is 0 and the program is terminated if any error has occurred. The termination level 1 would allow to continue if the error is not fatal (warning) and level 2 would allow to continue in any case. This could be useful for debugging a program. An occurrence of error level 2 usually means that the data structure is destroyed or likely to be destroyed and such a termination level should never be set for a running program or a production version.

#### 4.5.2 MOPS return codes

All MOPS system calls return an error code which is stored in a variable ERRNO and after successful execution of a call this value should be zero. Any other value indicates problems and a complete list of these error conditions can be found in appendix B.

If a programs stops, a line

```
FORTRAN STOP  90nn
```

is printed on the standard output, where "nn" is the error code of the termination subroutine.
4.5.3 Hexadecimal dump of a data structure

Sometimes a ASCII dump is not sufficient to investigate a data structure, especially when the data structure has been corrupted. A hexadecimal dump can be obtained with:

```plaintext
SUBROUTINE SDDUMP(Q)

INTEGER*4     Q          /* Address of first data structure */
```

A sorted hexadecimal dump is produced with this routine even if the data structure is corrupted or partially destroyed. An ASCII dump with QTOASC() would require a valid data structure.

4.5.4 Validity check

It is possible to test the validity of the data structure by a call to:

```plaintext
SUBROUTINE SDCHCK(MODE, Q)

INTEGER*4     MODE       /* Type of check to be performed (1 or 2) */
INTEGER*4     Q          /* Address of first data structure */
```

To check the directory only (normally sufficient) the MODE should be set to 1. If the mode is set to 2, the system will try to examine every data element in the data structure and test its validity and type. This takes very long and must not be a routine operation.

4.5.5 Copying a data structure

To copy a data structure into another dynamic store a routine is available:

```plaintext
SUBROUTINE SDCOPY(Q, II, NCOPY)

INTEGER*4     Q          /* Address of first data structure */
INTEGER*4     II         /* Address of second data structure */
INTEGER*4     NCOPY      /* Number of bytes copied */
```

The number of bytes copied into the second dynamic store is returned by this call in the variable NCOPY.

4.5.6 Comparing two data structures

To test the difference between two data structures a system call has been provided:
SUBROUTINE SDCOMP(Q, H, NBYTES)

INTEGER*4 Q / Star Address of first data structure */
INTEGER*4 H / Star Address of second data structure */
INTEGER*4 NBYTES / Star Number of bytes found different */

The number of bytes found different in the two data structures is returned by this call in the variable NBYTES.

4.6 Building relational data structures

In the MOPS data structure management system, a restricted possibility to build relational data structures has been implemented. Every object can have pointers to other data objects which are logically related to it. We have foreseen a limited number of six such "links" where one link is reserved to point to an object of the same type (horizontal link, linear data structure) and the other five links can be used to build a logical tree structure (vertical link, for naming conventions see ref. [5]). In those cases where the use of relational data structures is very important, it might be necessary to use other data structure management systems, which have been designed for interrelated data structures and for high efficiency at execution time (e.g. [5]).

An object can be associated with an already existing object if the call:

SUBROUTINE SDLIFT (NELEMS, ELEMSZ, NAME, TYPE, LINK, Q)

INTEGER*4 NELEMS / Star Number of elements */
INTEGER*4 ELEMSZ / Star Size of one element in bytes */
CHARACTER*80 NAME / Star Name of the data object */
CHARACTER*40 TYPE / Star Type of data */
INTEGER*4 LINK / Star Link number */
INTEGER*4 Q / Star Address of dynamic store */

is used instead of calls to SDBOOK() or SDZEBRO(). The link number LINK can have values between 0 and 5 where 0 would be used to create a link to an object of the same type (horizontal structure).

The call:

SUBROUTINE SDNEXT(NAME, POINT, Q)

CHARACTER*80 NAME / Star Name of user object */
INTEGER*4 POINT / Star Pointer into data structure */
INTEGER*4 Q / Star Address of dynamic store */

would return not the pointer to the object NAME, but to the next object of the same type as NAME in a horizontal structure. After the successful execution of this call, the name of the found object is returned in NAME. Successive calls to SDNEXT() would examine the complete horizontal structure associated with the object NAME until no more objects can be found and the pointer POINT would return −1.
Access to vertical data structures (i.e. LINK .NE. 0) is obtained by:

```
SUBROUTINE SDCHILD(NAME, POINT, LINK, Q)

CHARACTER*80 NAME /* Name of user object */
INTEGER*4 POINT /* Pointer into data structure */
INTEGER*4 LINK /* Link to associated object (1 - 5) */
INTEGER*4 Q /* Address of dynamic store */
```

All links are updated when a garbage collection occurs or when the size of an object is changed. However, links to an object dependent on a dropped object remain valid until a garbage collection is executed.

### 4.7 Network transfer

The exchange of MOPS data structures between computers with a different data representation is possible using the MOPS exchange format [7]. To allow the exchange of objects with arbitrary data elements like "C" structures, the contents of such a structure has to be made available to the MOPS system. This can be done with a call to:

```
SUBROUTINE SDFORM(NAME, STRING, Q)

CHARACTER*80 NAME /* Name of the structure, e.q. 'plot' */
CHARACTER*80 STRING /* Structure description */
INTEGER*4 Q /* Address of dynamic store */
```

The description of the structure is provided in a string like\(^7\) '3. FLOAT, 12, CHAR, 3, INT, 1024, FLOAT, 2, INT' which would describe a structure like:

```c
struct background {
    float xy_dat[3];
    char name[12];
    int setting[3];
    float data[1024];
    int shot[2];
}
```

and is used as:

```c
string = '3,FLOAT,12,CHAR,3,INT,1024,FLOAT,2,INT'"
CALL SDFORM('background', STRING, Q)
```

This call will encode the format string into a variable number of words describing the structure and store it in the data structure. It must be called before this structure can be used in a call to SDBOOK() or SDZERO(). In SDBOOK() or SDZERO() this structure would be referred to as the

\(^7\) For compatibility the types should be in upper case although the "C" version would accept lower case and upper case types.
type "struct background". Since a system object is generated for every structure, the user must increase
the number of objects initialised with SDINI() to accommodate this system object. A call to
SDFORM() for a structure is only necessary if the data structure is send across the network with the
mopsserver [7] otherwise it is sufficient to book the object with SDBOOK() or SDZERO().

5. RUNNING INSTRUCTIONS

The MOPS data structure management system has been installed on IBM VM, APOLLO work-
stations, VAX VMS and on NORD 500 running the SINTRAN operating system. In this chapter we
would like to give an overview what is needed to use MOPS on these machines and how to install it.
It is usually sufficient to obtain access to two files: the object library with the MOPS support programs
and the include file which has to be included into all subroutines which need access to MOPS variables
like NSTOP or the directory variables.

5.1 IBM VM/CMS

To use MOPS on CERN IBM VM the disk

   GIME ZWE 191

has to be accessed which contains the library

   MOPSLIB TXTLIB

and a file to be included into programs which use MOPS system calls:

   MOPS INCL

The source code is available in PATCHY format [6] as a PAM file or as preprocessed FORTRAN 77
source code and a MAKEFILE is provided to perform the installation procedure. The most recent
source code for the MOPS system is always stored on CERNVM and can be accessed from all other
installations via TCP/IP.

5.2 NORD 500 SINTRAN

Currently the MOPS system is installed on the SPS modelling machine (MODEL) and the library and
include file are:

   (MODEL-UTILITY)MOPS-UTILITY-LIB:NRF

   (MODEL-UTILITY)MOPS-UTILITY:INCL

For the installation a mode file is provided. The source code is available on the NORD 500 disk or
can be taken from CERNVM if desired.
5.3 APOLLO workstations

On APOLLO workstations the MOPS system has been installed on the SPS DOMAIN and token—ring APOLLO network and can be used with:

/usr/lib/mopslib.a
/usr/include/mops.incl

A makefile is provided which can be used to install it on any APOLLO workstation ( /user/acteam/mops/makemaps ). The source code is always taken from CERNVM when this makefile is executed. An additional library and an include file must be specified in order to use the shared memory option in the UNIX environment of the APOLLO [3]:

/usr/lib/libshm.a
/usr/include/sps.shm.h

5.4 VAX VMS

The library and include file are available on DISK$SI: [ZWI] as:

MOPS.OLB
MOPS.INCL

An automatic installation procedure exists which can be used to install MOPS on any computer or account ( INSTALL.COM ). The source code is automatically taken from CERNVM by this procedure.

5.5 Technical details

The user can get access to the information stored in the directory using the system calls described above (i.e. SDFIND and SDGREP). Therefore an include file must be included in every program which executes these programs. The directory information is copied into a common block:

COMMON/DIREC/ OFFSET, NELEMS, ELEMMSZ, OBJNAM, ELEMCD, ELTYPE, PLOT &
   , L00, L01, L02, L03, L04, L05

declared as:

INTEGER*4 OFFSET, NELEMS, ELEMMSZ, ELEMCD, PLOT
INTEGER*4 L00, L01, L02, L03, L04, L05
CHARACTER*80 OBJNAM
CHARACTER*40 ELTYPE

The directory can therefore be seen as a sequence of such common blocks. Since FORTRAN 77 does not support structured datasets the user has no direct access to directory entries unless our utilities are used.
The meaning of these variables is as follows:

**OFFSET**: Offset to the beginning of the user object in bytes

**NELEMS**: Number of elements in the object

**ELEMSZ**: Size of one element in bytes

**OBJNAM**: Name of the object (keyword)

**ELEMCD**: Element code (used internally only, NODAL like types)

**ELTYPE**: Element type as described in section 3.2

**PLOT**: Plot index used by SDPLOT and SDHIST

**L00 - L05**: Structural links

The subroutine SDSTAT uses another common block to communicate with the user programs.

The status is copied into a COMMON block

```
COMMON/SDTTC/, SHLEN, SIINAME, SHNEL
```

where:

**INTEGER**+4 SHLEN, SHNEL

**CHARACTER**+80 SIINAME

**SHLEN**: Total length of the data structure in bytes

**SHNEL**: Number of objects currently initialised

**SHNAM**: Name of the data structure
6. APPENDIX A

EXAMPLE

PROGRAM GOOF

INCLUDE 'MOPS.INCL'

REAL*4 II(144), KL(144), A(144), D(144)

C -- integer array Q as dynamic store ...

INTEGER*4 Q(100000)
CHARACTER*20 NAME1, NAME2, NAME3, NAME4, NAME
CHARACTER*20 ITYPE, RTYPE, CTYPE

C -- define types for later use

ITYPE = 'INTEGER*4'
RTYPE = 'REAL*4'
CTYPE = 'CHARACTER'

C -- initialise for 15 user objects

NAME = 'TWISS'
CALL SDINI(NAME, 15, Q)

C -- read data from file 15 and 16

READ(15) A,D
READ(16) II,KL

C -- initialise for 2 real objects

NAME1 = 'BETA1'
NAME2 = 'BETA2'

CALL SDBOOK(144, 4, NAME1, RTYPE, Q)
CALL SDBOOK(144, 4, NAME2, RTYPE, Q)

C -- fill first 2 real objects with data
C -- 1.: fill 144 elements of KL into 2nd object

CALL SDWRIT(NAME2, 108, KL, Q)

C -- 2.: fill 108 elements of II into 1st object

CALL SDWRIT(NAME1, 108, II, Q)

C -- initialise more objects

NAME3 = 'ALFA1'
NAME4 = 'ALFA2'

CALL SDBOOK(144, 4, NAME3, RTYPE, Q)
CALL SDBOOK(144, 4, NAME4, RTYPE, Q)

C - - - fill A and D into data structure

CALL SDWRIT(NAME3, 114, A, Q)
CALL SDWRIT(NAME4, 114, D, Q)

C - - - print data from data structure

CALL SHOW(Q)

C - - - write entire data structure to file "TWISS"

CALL SDFIL("TWISS",Q)

STOP 555
END

SUBROUTINE SHOW(F)

INCLUDE 'MOPS.INCL,'

INTEGER*4 POINT1, POINT2, F(1)
REAL*4 FR(1), A(20), B(20)
CHARACTER*20 NAME1, NAME2, OBINAM(2)
EQUIVALENCE (F(1), FR(1))
OBINAM(1) = 'BETAX'
OBINAM(2) = 'BETAY'

C - - - get access to $\beta$ functions stored in data structure

CALL SDPTR(OBINAM(1), POINT1, F)
CALL SDPTR(OBINAM(2), POINT2, F)
DO 301 K = 1, 144
   J = K - 1
301 PRINT 6001, FR(POINT1 + J), FR(POINT2 + J)
6001 FORMAT(1H1, 2X;'BETAX: ',F12.5,4X,'BETAY: ',F12.5)

C - - - real objects / copy to local array first

NAME1 = 'ALFAX'
NAME2 = 'ALFAY'
CALL SDREAC(NAME1, POINT1, A, F)
CALL SDREAC(NAME2, POINT2, B, F)

PRINT 6002, (A(I), B(I), I = 1, 108)
6002 FORMAT(1H1, 3X, F10.1, 15X, F10.5)

RETURN
END
# 7. APPENDIX B

## MOPS ERROR CODES

<table>
<thead>
<tr>
<th>Code</th>
<th>Level</th>
<th>Routine</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>SDINI</td>
<td>Data structure has no elements</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>SDBOOK</td>
<td>No more space in data structure</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>SDBOOK</td>
<td>Inconsistent element size</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>SDBOOK</td>
<td>String array has too many bytes (NORD only)</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>SBOOK</td>
<td>Offsets clobbered</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>SDWRT</td>
<td>Object not found in data structure</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>SDWRT</td>
<td>Inconsistent parameters in calling sequence</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>SDREAD</td>
<td>Object not found in data structure (obsolete)</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>SDREAD</td>
<td>Object not correctly initialised (obsolete)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>SREAC</td>
<td>Object not found in data structure</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>SREAC</td>
<td>Object not correctly initialised</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>SREAX</td>
<td>Object not correctly initialised</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>SREAX</td>
<td>Sequence number too large</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>SREAY</td>
<td>Object not correctly initialised</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>SREAY</td>
<td>Sequence number too large</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>SDFIND</td>
<td>Object not found in directory</td>
</tr>
<tr>
<td>17</td>
<td>2</td>
<td>SDFIND</td>
<td>Object not correctly initialised</td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>SDGREP</td>
<td>Object not correctly initialised</td>
</tr>
<tr>
<td>19</td>
<td>2</td>
<td>SDGREP</td>
<td>Sequence number too large</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>SDFIL</td>
<td>File cannot be opened</td>
</tr>
<tr>
<td>21</td>
<td>2</td>
<td>SDFIL</td>
<td>File cannot be opened</td>
</tr>
<tr>
<td>22</td>
<td>2</td>
<td>SDSTAT</td>
<td>Not a MOPS data structure</td>
</tr>
<tr>
<td>23</td>
<td>2</td>
<td>SDSTAT</td>
<td>Cannot open file for hexadecimal dump</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td>SGAR   B</td>
<td>No dropped objects found, garbage collection terminated</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>SDDUMP</td>
<td>Cannot open file for hexadecimal dump</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td>SDLIFT</td>
<td>No more space in data structure</td>
</tr>
<tr>
<td>27</td>
<td>1</td>
<td>SDLIFT</td>
<td>Inconsistent element size</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>SDLIFT</td>
<td>String array has too many bytes (NORD only)</td>
</tr>
<tr>
<td>29</td>
<td>2</td>
<td>SDLIFT</td>
<td>Offsets clobbered</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>SDNEXT</td>
<td>Object not found in data structure</td>
</tr>
<tr>
<td>31</td>
<td>2</td>
<td>SDNEXT</td>
<td>Object not correctly initialised</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>SDCIII.D</td>
<td>Object not found in data structure</td>
</tr>
<tr>
<td>33</td>
<td>2</td>
<td>SDCIII.D</td>
<td>Object not correctly initialised</td>
</tr>
<tr>
<td>34</td>
<td>1</td>
<td>SDCIII.C</td>
<td>Object not found in data structure</td>
</tr>
<tr>
<td>35</td>
<td>2</td>
<td>SDCIII.C</td>
<td>Invalid link number</td>
</tr>
<tr>
<td>36</td>
<td>2</td>
<td>SDCIII.C</td>
<td>Object not correctly initialised</td>
</tr>
<tr>
<td>37</td>
<td>1</td>
<td>SDDRAW</td>
<td>No plot found in plot object</td>
</tr>
<tr>
<td>38</td>
<td>1</td>
<td>SDDAW</td>
<td>No plot found in plot object</td>
</tr>
<tr>
<td>39</td>
<td>2</td>
<td>SDDAW</td>
<td>Cannot open PAW output file</td>
</tr>
<tr>
<td>40</td>
<td>2</td>
<td>ASCTIQ</td>
<td>Error reading ASCII input file</td>
</tr>
<tr>
<td>41</td>
<td>2</td>
<td>QTASC</td>
<td>Error writing to ASCII output file</td>
</tr>
<tr>
<td>42</td>
<td>2</td>
<td>SBOOK</td>
<td>Element type not allowed (MINIMOPS/VME only)</td>
</tr>
<tr>
<td>43</td>
<td>1</td>
<td>SDLTDF</td>
<td>Invalid plot type found, request ignored</td>
</tr>
<tr>
<td>44</td>
<td>1</td>
<td>SDDAW</td>
<td>Invalid plot type found, request ignored</td>
</tr>
<tr>
<td>45</td>
<td>2</td>
<td>SBOOK</td>
<td>Invalid data type</td>
</tr>
<tr>
<td>Code</td>
<td>Level</td>
<td>Routine</td>
<td>Error</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>---------</td>
<td>-----------------------------------------------------------</td>
</tr>
<tr>
<td>46</td>
<td>2</td>
<td>SDLFHT</td>
<td>Invalid data type</td>
</tr>
<tr>
<td>47</td>
<td>1</td>
<td>SDPTR</td>
<td>Object not found in data structure</td>
</tr>
<tr>
<td>48</td>
<td>2</td>
<td>SDPTR</td>
<td>Object not correctly initialised</td>
</tr>
<tr>
<td>49</td>
<td>2</td>
<td>SDLFHT</td>
<td>Cannot link to object</td>
</tr>
<tr>
<td>50</td>
<td>2</td>
<td>QTOINF</td>
<td>Error writing to internal file</td>
</tr>
<tr>
<td>51</td>
<td>2</td>
<td>SDCUT</td>
<td>Requested length bigger than originally booked</td>
</tr>
<tr>
<td>52</td>
<td>2</td>
<td>SDCUT</td>
<td>Object is not correctly initialised</td>
</tr>
<tr>
<td>53</td>
<td>1</td>
<td>SDCUT</td>
<td>Object not found in data structure</td>
</tr>
<tr>
<td>54</td>
<td>2</td>
<td>SDAUGM</td>
<td>Requested length smaller than originally booked</td>
</tr>
<tr>
<td>55</td>
<td>2</td>
<td>SDAUGM</td>
<td>Object is not correctly initialised</td>
</tr>
<tr>
<td>56</td>
<td>1</td>
<td>SDAUGM</td>
<td>Object not found in data structure</td>
</tr>
<tr>
<td>57</td>
<td>2</td>
<td>SDSTO</td>
<td>File cannot be opened</td>
</tr>
<tr>
<td>58</td>
<td>2</td>
<td>SDUSTO</td>
<td>File cannot be opened</td>
</tr>
<tr>
<td>59</td>
<td>1</td>
<td>SDZERO</td>
<td>No more space in data structure</td>
</tr>
<tr>
<td>60</td>
<td>1</td>
<td>SDZERO</td>
<td>Inconsistent element size</td>
</tr>
<tr>
<td>61</td>
<td>1</td>
<td>SDZERO</td>
<td>String array has too many bytes (NORD only)</td>
</tr>
<tr>
<td>62</td>
<td>2</td>
<td>SDZERO</td>
<td>Offsets clobbered</td>
</tr>
<tr>
<td>63</td>
<td>2</td>
<td>SDZERO</td>
<td>Element type not allowed (MINIMOPS/VME only)</td>
</tr>
<tr>
<td>64</td>
<td>2</td>
<td>SDZERO</td>
<td>Invalid data type</td>
</tr>
<tr>
<td>65</td>
<td>2</td>
<td>SDDPUT</td>
<td>Not a UNIX system</td>
</tr>
<tr>
<td>66</td>
<td>2</td>
<td>SDDGET</td>
<td>Not a UNIX system</td>
</tr>
<tr>
<td>67</td>
<td>2</td>
<td>SDCCHCK</td>
<td>Not a MOPS data structure</td>
</tr>
<tr>
<td>68</td>
<td>1</td>
<td>SDCCHCK</td>
<td>Not a valid mode</td>
</tr>
<tr>
<td>69</td>
<td>1</td>
<td>SDCCHCK</td>
<td>Type cannot be tested on this computer</td>
</tr>
<tr>
<td>70</td>
<td>1</td>
<td>SDCCHCK</td>
<td>Invalid element size</td>
</tr>
<tr>
<td>71</td>
<td>2</td>
<td>SDCCHCK</td>
<td>Offsets clobbered</td>
</tr>
<tr>
<td>72</td>
<td>1</td>
<td>SDCCHCK</td>
<td>Data in object of wrong type</td>
</tr>
<tr>
<td>73</td>
<td>2</td>
<td>SDCCHCK</td>
<td>Data buffer to small for REAL*8 data</td>
</tr>
<tr>
<td>74</td>
<td>1</td>
<td>SDGKS</td>
<td>Object not found</td>
</tr>
<tr>
<td>75</td>
<td>1</td>
<td>SDGKS</td>
<td>No plots found in plot structure</td>
</tr>
<tr>
<td>76</td>
<td>1</td>
<td>SDGKS</td>
<td>Invalid plottype</td>
</tr>
<tr>
<td>77</td>
<td>1</td>
<td>SDDRAW</td>
<td>Invalid datatype</td>
</tr>
<tr>
<td>78</td>
<td>1</td>
<td>SDHIPO</td>
<td>No plots found in plot structure</td>
</tr>
<tr>
<td>79</td>
<td>1</td>
<td>SDHIPO</td>
<td>Invalid plottype</td>
</tr>
<tr>
<td>80</td>
<td>1</td>
<td>SDHIPO</td>
<td>Invalid data type</td>
</tr>
<tr>
<td>81</td>
<td>1</td>
<td>SDHIPO</td>
<td>Invalid workstation type</td>
</tr>
<tr>
<td>82</td>
<td>1</td>
<td>SDMETA</td>
<td>No plots found in plot structure</td>
</tr>
<tr>
<td>83</td>
<td>1</td>
<td>SDMETA</td>
<td>Invalid plottype</td>
</tr>
<tr>
<td>84</td>
<td>1</td>
<td>SDMETA</td>
<td>Invalid data type</td>
</tr>
<tr>
<td>85</td>
<td>1</td>
<td>SDMETA</td>
<td>Invalid workstation type</td>
</tr>
<tr>
<td>86</td>
<td>1</td>
<td>SDSNAP</td>
<td>More elements to copy than initialised</td>
</tr>
<tr>
<td>87</td>
<td>1</td>
<td>SDSNAP</td>
<td>Object not found in data structure</td>
</tr>
<tr>
<td>88</td>
<td>2</td>
<td>SDSNAP</td>
<td>Object not correctly initialised</td>
</tr>
<tr>
<td>89</td>
<td>1</td>
<td>SDSPOKE</td>
<td>More elements to copy than initialised</td>
</tr>
<tr>
<td>90</td>
<td>1</td>
<td>SDSPOKE</td>
<td>Object not found in data structure</td>
</tr>
<tr>
<td>91</td>
<td>2</td>
<td>SDSPOKE</td>
<td>Object not correctly initialised</td>
</tr>
<tr>
<td>Code</td>
<td>Level</td>
<td>Routine</td>
<td>Error</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>---------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>92</td>
<td>1</td>
<td>SDFORM</td>
<td>Object name invalid</td>
</tr>
<tr>
<td>93</td>
<td>2</td>
<td>SDFORM</td>
<td>Invalid format descriptor</td>
</tr>
<tr>
<td>94</td>
<td>2</td>
<td>SDPARS</td>
<td>Invalid data type in format descriptor</td>
</tr>
<tr>
<td>95</td>
<td>2</td>
<td>SDPARS</td>
<td>Invalid format descriptor</td>
</tr>
</tbody>
</table>
8. REFERENCES


Acknowledgments

We would like to express our thanks to those people who have contributed in the design and development of the MOPS data structure management system. In particular we would like to thank Janice Reid, who has written a large fraction of the first FORTRAN 77 implementation and Chris Saltmarsh who has participated in the design stage. For many valuable discussions and suggestions we would like to thank our colleagues K. Cornelis, A. Hilaire, R. Lauckner and G. Morpurgo.
## INDEX OF MOPS SUBROUTINES:

<table>
<thead>
<tr>
<th>Subroutine</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCTOQ</td>
<td>9</td>
</tr>
<tr>
<td>QTOASC</td>
<td>9</td>
</tr>
<tr>
<td>QTOINF</td>
<td>9</td>
</tr>
<tr>
<td>SDAUGM</td>
<td>11</td>
</tr>
<tr>
<td>SDBOOK</td>
<td>4</td>
</tr>
<tr>
<td>SDCIICK</td>
<td>16</td>
</tr>
<tr>
<td>SDCHLD</td>
<td>18</td>
</tr>
<tr>
<td>SDCOMP</td>
<td>17</td>
</tr>
<tr>
<td>SDCOPY</td>
<td>16</td>
</tr>
<tr>
<td>SDCUT</td>
<td>11</td>
</tr>
<tr>
<td>SDDRAW</td>
<td>14</td>
</tr>
<tr>
<td>SDDROP</td>
<td>10</td>
</tr>
<tr>
<td>SDDUMP</td>
<td>16</td>
</tr>
<tr>
<td>SDERR</td>
<td>15</td>
</tr>
<tr>
<td>SDFIL</td>
<td>8</td>
</tr>
<tr>
<td>SDFIND</td>
<td>12</td>
</tr>
<tr>
<td>SDFORM</td>
<td>18</td>
</tr>
<tr>
<td>SDGARB</td>
<td>10</td>
</tr>
<tr>
<td>SDGET</td>
<td>10</td>
</tr>
<tr>
<td>SDGREP</td>
<td>12</td>
</tr>
<tr>
<td>SDHIST</td>
<td>13</td>
</tr>
<tr>
<td>SDHPLOE</td>
<td>14</td>
</tr>
<tr>
<td>SDINI</td>
<td>3</td>
</tr>
<tr>
<td>SDLEN</td>
<td>13</td>
</tr>
<tr>
<td>SDLIFT</td>
<td>17</td>
</tr>
<tr>
<td>SDMETA</td>
<td>14</td>
</tr>
<tr>
<td>SDNEXT</td>
<td>17</td>
</tr>
<tr>
<td>SDPAW</td>
<td>15</td>
</tr>
<tr>
<td>SDPLOT</td>
<td>13</td>
</tr>
<tr>
<td>SDPLTC</td>
<td>14</td>
</tr>
<tr>
<td>SDPOKE</td>
<td>6</td>
</tr>
<tr>
<td>SDPTR</td>
<td>6</td>
</tr>
<tr>
<td>SDPUT</td>
<td>10</td>
</tr>
<tr>
<td>SDREAD</td>
<td>7</td>
</tr>
<tr>
<td>SDREADX</td>
<td>12</td>
</tr>
<tr>
<td>SDREADY</td>
<td>12</td>
</tr>
<tr>
<td>SDSNAP</td>
<td>7</td>
</tr>
<tr>
<td>SDSTAT</td>
<td>13</td>
</tr>
<tr>
<td>SDSTO</td>
<td>9</td>
</tr>
<tr>
<td>SDUHL</td>
<td>8</td>
</tr>
<tr>
<td>SDUSTO</td>
<td>9</td>
</tr>
<tr>
<td>SDWRIT</td>
<td>5</td>
</tr>
<tr>
<td>SDZERO</td>
<td>5</td>
</tr>
</tbody>
</table>
GENERAL INDEX

APOLLO ... 20
ASCII output ... 9

basic concept ... 1
binary output ... 9
booking ... 4

checking ... 16
comparison ... 17
copying ... 16

data access ... 6
data conversion ... 9
data locking ... 7
data storage ... 5
data structure I/O ... 8
data structure status ... 13
dataviewer ... 13
directory ... 1
directory access ... 8
directory common ... 12
directory structure ... 20
directory variables ... 21
dropping ... 10
dumping ... 16
dynamic store ... 1, 3

element ... 1
element code ... 2
element size ... 2
element type ... 2
element types ... 4
error codes ... 15
error conditions ... 15
error level ... 15
error messages ... 15
error output ... 15
exchange format ... 18
external medium ... 8

format list ... 18

garbage collection ... 10
GKS ... 14
GKS metatile ... 14

HBOOK ... 14
hexadecimal dump ... 16
histogramming ... 13
horizontal link ... 17
HPILOT ... 14

IBM VM/CMS ... 19
initialisation ... 3
installation procedure ... 19
internal file ... 9

keyword ... 2

linear structure ... 17
logical unit number ... 9

makefile ... 19
metatile ... 14
metatile printout ... 14
metatile transfer ... 14
MOPS ... 1
MOPS include file ... 8, 19
MOPS library ... 19
MOPS return codes ... 15

normalised pointers ... 6
NORSK DATA ... 19
number of elements ... 2

object ... 1
object name ... 2
occupied space ... 10
offset ... 2

PAW ... 15
plot object ... 13
plotindex ... 13
plotting ... 13
pointer ... 5
portability ... 3

random access ... 8
relational structure ... 17
running instructions ... 19

scatter plots ... 14
sequence number ... 12
sequential access ... 8
shared memory ... 9, 20
shared memory name ... 9
size change ... 11
string descriptor ... 7
system object ... 1

termination level ... 15
total length ... 13
tree structure ... 17
type check ... 7
UNIX ... 9
user object ... 1
valid types ... 2, 4
validity check ... 13
VAX VMS ... 20
vertical link ... 17