DSPACK

User's Guide
(Version 1.010, preliminary)

Ryszard Zybert
School of Physics and Space Research
University of Birmingham

5^{th} December 1993

Acknowledgements

Many ideas used in DSPACK originate from SYPACK, data management package written for the NA36 experiment by Gregg Franklin. Many people contributed to SYPACK in many stages and many of their ideas and solutions found their way to the design of DSPACK. I would like to acknowledge their contribution here.

DSPACK would probably never see the light of day without the help of Predrag Buncic who proved to me that a good programmer does not need any manuals, help files or instructions.

1 Introduction

A large software package as that needed to process data from the NA49 experiment requires a data handling system. As different parts of the package will be written by different people, in different languages - the data handling system must supply a 'glue' which will help to keep the package together. The system must be fast as the volume of data is expected to be very large.

The data handling package must not limit access from any language and must be flexible enough to allow future modifications in ways that we cannot even see right now.

CERN's package ZEBRA provides a powerful tool that could be used to construct a data handling package. Although ZEBRA is written in FORTRAN and is designed to be used by FORTRAN programs - the data handling system based on ZEBRA does not have to have this limitation. The important advantage of using ZEBRA as an underlying memory management system is that it gives access to a multitude of current and future tools of CERN library that are designed around the ZEBRA system.

NA36 developed its own data handling system SYPACK that did not have many of limitations of ZEBRA (replacing them by its own limitations that were considered irrelevant for the NA36 experiment). SYPACK does its own memory management - this means that conversion tools are needed in order to use many of the CERN library tools. As the result a set of tools has been developed for NA36 based on direct access to SYPACK data. In addition a limited number of conversion tools have been produced in order to use CERN library tools with NA36 data.

In DSPACK the data management and memory management are separated. It follows SYPACK philosophy but it may use different memory managers in a way transparent to the user. At the same time DSPACK goes much further in trying to accommodate modern programming techniques - like structures and pointers - as well as preserving the convenience of creating objects dynamically and accessing them in a fast and convenient way.

2 DSPACK features

- DSPACK provides a simple technique of passing data between the program and the data manager. The interface is identical for fortran and c programs and is designed to accommodate different programming styles.
- All objects are defined dynamically, at execution time. There are many advantages of such an arrangement: object definition may be made dependent on the run-time options or parameters, in interactive programs objects may be created by the user, the histogramming and test/cut packages may create their own objects, etc.
- All objects are independent and complete - this means that any reference to an object implies a reference to everything this object depends on. Copying, moving or writing an object to disk or tape produces an entity that is functionally identical to the original. (At least this was the intention and this is the present experience and the author is determined to keep it that way)
- Objects are addressed by a name of up to 16 characters and a handle (pointer to a pointer). The correct handle is returned by all routines accessing data after a successful search for the name. Provided that the programmer does not modify the handle the name is not used in subsequent calls and objects are found using the handle - even if they were re-located in the mean-time. Additional protection is provided so that the unsuccessful search is not repeated if it is bound to fail (if no new objects were defined since the last failure). The same method is used to access elements of the object - including individual words. This technique provides easy access to data by name (making the programs much more readable too) and at the same time is extremely fast - only two levels of indirection and no searches are needed to access any object after the first call. Addressing exclusively by name is used by interactive programs where speed is not important - the search never takes enough time to be noticed by the user.
- DSPACK output files and tapes are 'self-defining'. The header of a dspack file or tape contains complete definitions of all objects that may be found in the file - a dspack program defines all needed objects after reading the header. This means that any dspack file or tape may be read by any dspack program. No prior knowledge of the tape contents is necessary. Nevertheless - the dspack library contains tools which can read a header of a dspack file or tape and produce f77 or c header files defining all the objects found.
- If used with ZEBRA as a memory manager DSPACK uses FZ package for I/O. This provides the flexibility of FZ and at the same time allows dspack files to be read directly by other packages - like PAW. This way dspack objects may be accessed and analysed by PAW or other programs using ZEBRA.
- DSPACK data sets are 'self-documenting'. Each data set has a corresponding definition record in the permanent division containing some amount of documentation. Some of that information is used by DSPACK tools, other is provided for documentation and help in interactive work.

- Access to data is either through routines which copy data (GET and PUT) or through pointers to objects or to individual words. The pointers are returned by DSPACK functions called with data set name and handle.

- DSPACK may be used in a local or server-client mode . In the server-client mode the space is shared between several clients which may access data as usual, however the dspack calls which store data are automatically passed to a DSPACK deamon which manages the memory. Also all dspack calls dealing with I/O are passed for execution to the deamon. The dspack deamon may control access to data although the system is in principle designed to have only one process modifying data at any given time. The selection of local or server-client mode is made at execution time when DSPACK is initialized - after this the user's code is completely independent of mode - the necessary communication with the server is done internally by DSPACK.

- DSPACK has been designed to provide complete isolation of production programs from the memory handling software. This is achieved by applying a very strict rule which does not allow any communication with DSPACK other than by calling DSPACK routines (there are no global variables or common blocks available to the calling program). Important advantage of that approach is that the production programs do not need to change at all whatever changes are made to the memory management system or I/O format. In the present form DSPACK may be replaced by any program as long as it conserves the DSPACK's calling sequences.

3 Basic concepts

DSPACK data set is a two-dimensional object (very much like an array of structures). It is defined by the program - usually in initialization phase (although new objects may be defined at any time - for instance when reading a dspack tape) and this definition is used later when the data set is used. The data set definition specifies the number of items (corresponding to elements of a structure), the number of entries (the count of how many times the structure is repeated) is 'dynamic' - it may change every time data are stored in the data set. Access to the whole data set or individual entries as well as to individual words is possible through DSPACK calls. DSPACK items may have different length - they may even be arrays of structures themselves.

Data set is identified by data set name (up to 16 characters). However - for efficiency, all DSPACK calls accessing data do it using two arguments: data set name and data set handle (address of the data set pointer). The search for the data set based on the name is performed in the first call which then returns the correct value of the handle. As long as the user does not modify the handle all subsequent calls address the data directly, at most checking whether the handle and the name agree (although even this test may be disabled by the caller) - even if the actual address of the data set changes in the meantime - for instance in the new event. This method provides fast access to data where necessary but allows access by name if this is more convenient - for instance in interactive programs the user enters data set names and the full search is performed every time (all pointers and names are kept in one place which makes the search quite fast anyway).

Example:

Fortran:

```
save my_handle
data my_handle/0/

...
...
call dsget('MY_DATA_SET',my_handle....
```

c:

```
static int my_handle=0;
...
...
dsget("MY_DATA_SET",&my_handle...
```

In both cases the name 'MY_DATA_SET' is only used on the first call. On all subsequent calls variable my_handle provides quick access to the data even if the data set has been removed and stored again, for instance in the next event. The programmer must have a separate variable to store the handle for each data set accessed and the values returned by DSPACK should never be modified.

Each item of a data set has a name (up to 16 characters). This is mainly used by tools (like histogram definitions, calculator, test package) and interactive programs but may also be used in calls accessing data. Additionally each item may have an additional 20 character description which is used by interactive programs as a comment. A multi-entry data set fills a contiguous region in memory . Therefore it is usually filled completely by one call or the memory is 'booked' and then filled entry-by-entry. Adding entries to a complete data set is expensive. However a mechanism is provided to fill a data set of unknown size and to declare the size afterwards - thus avoiding the need to know the number of entries in advance.

If used with ZEBRA memory manager DSPACK I/O uses ZEBRA FZ I/O routines. However it defines a convention of how the data should be written in order to make sure that all information needed by DSPACK is kept. The user makes a list of data sets to be written out after each event. All the definitions of all data sets tagged this way are written as a first block of DSPACK output. After each event all tagged data sets (and their dependants) are written out. (The same system is applied separately to constants stored by DSPACK). When the DSPACK file is read by DSPACK I/O, the header with definitions is used to construct the definitions for all data sets in the file. This means that the reading program does not need to define any structures which it reads from the file or tape. Of course the FZ files may be read by any program using ZEBRA (like PAW) and used directly.

3.1 Data set attributes

Each data set has to be declared to DSPACK with a list of attributes which describe it. Following attributes are required:

NAME - character identifier of the data set (up to 16 characters, no dots allowed). DSPACK is case sensitive but accepting a convention of using only upper case makes the interactive work easier.

TYPE - following data set types are allowed at present:

1	'I'	integer	4 bytes
2	'R'	real	4 bytes
3	'L'	logical	4 bytes
4	'D'	double precision	8 bytes
5	'B'	bit string	4 bytes
6	'C*n'	fixed character	n bytes (mod(n,4)=0)
7	'C'	character	dynamic
8		mixed	definition dependent
9	'S'	structure	definition dependent

SECTION - dspack stores separately constants and variables. Variables are kept in section 1 and will usually be cleared after each event. Constants are kept in section 2 and are permanent. Section 0 means that the data set cannot hold data, the definition is only a template to be used in other definitions.

NUMBER OF ITEMS

ITEM NAMES - each item has a name up to 16 characters.

For structures each item also requires the name of data set which describes the item.

More attributes (like object class) may be stored with the data set definition but are not required immediately when defining the data set.

3.2 Data set types

The first six types shown above describe simple data sets. They provide the best efficiency and should be used wherever possible. An integer data set with n items simply contains n integer numbers per entry etc. The complete data set is just an integer two-dimensional array with n rows and the number of columns equal to the number of entries. The number of rows is fixed, the number of columns may change from event to event.

The other types need more explanations:

Dynamic character data set These data sets have always a single entry. The data set may have a number of items all of type character and the length of items (one length, common for all items) is decided when the data are stored. Special routines are provided to store and retrieve data of this type.

Mixed data set The ability of saving data of different types into one data set may be important if some c-type structures are to be mapped into the dspack objects. Mixed data sets may contain any number of items of different types - the types 1 to 6 above are allowed. When defining a mixed data set the programmer specifies the content in a way very similar to that used by ZEBRA - with following differences:

- a) nL tokens are allowed for LOGICAL items
- b) no "/" or '*' or 'S' items are allowed

- c) nH token must be specified separately for each item, even if they follow each other ('3H 3H' rather than '6H' must be used to describe two 12-character items)

Mixed data sets are intended for short, simple definitions. For larger objects a much more efficient method is to use Structure type.

Structures These data sets consist of items which are defined data sets (or arrays of data sets) themselves. Each item may in turn consist of structures etc.

For example - imagine that we want to describe a box in space - like a detector. It can be built of 2 frames each having 4 corners which are just space points. Additionally we may want to keep another space point giving the position of the centre of the box and some flags needed to display the detector with the event display - like colour, on/off flag and the name.

The way we would do it in c would be something like that:

```
typedef struct point
{
               float x;
               float y;
               float z;
} point;

typedef struct frame
{
               point c[4];
} frame;

typedef struct bflags
{
               int colour;
               int onoff;
               char name[16];
} bflags;

typedef struct box
{
               frame f[2];
               point position;
               bflags flags;
} box;
```

In f77 on a UNIX machine or in VAX FORTRAN it would look like that:

```
        structure /POINT/
                  real  X
                  real  Y
                  real  Z
        end structure
```

```
structure /FRAME/
          record/POINT/  C(4)
end structure

structure /BFLAGS/
          Integer*4        COLOUR
          Logical*4        ONOFF
          Character*16     NAME
end structure

structure /BOX/
          record/FRAME/  F(2)
          record/POINT/  POSITION
          record/BFLAGS/  FLAGS
end structure
```

In both languages we may now create an array of boxes and store our data in it. This construct can be exactly matched by a dspack data set (in fact the text above was produced by a tool which reads a header of a dspack data file and outputs c or fortran headers describing the data sets found there).

Here is how this is defined in dspack:

In fortran:

```
* define POINT - REAL, 3 items (X,Y,Z), extra comments
* for interactive work ('|' separates items if they contain spaces)
* section 0 means that this object is only a template to be used
* in other definitions rather than to store data
* Using a non-zero section is allowed - POINT would then become
* a data set definition as well and one could store data in it.
* Section number is ignored when POINT is used in other
* definitions.


      call dsdef('POINT','R',0,3,
     >   'X|Y|Z|X coordinate|Y coordinate|Z coordinate',iv,ierr)


* Now define FRAME - a structure - an array of 4 points (1 item)


      call dsdef('FRAME','S',0,1,'C(4) POINT',iv,ierr)


* BFLAGS - to keep extra flags - 3 items - 1 integer,1 logical
*          and one 16 character string - here none of the words
```

```
*           contains spaces, so a space may be used for a separator
* (The comments 'Box_colour On_off Box_name' are optional but
* recommended)
* Note the second argument describing the data set type


      call dsdef('BFLAGS','1I 1L 4H',0,3,
     >                    'COLOUR ONOFF NAME '//
     >                    'Box_colour On_off Box_name',iv,ierr)


* Now we are ready to define our box - a structure with 3 items
* the first is an array of 2 frames F followed by a point POSITION
* and a structure FLAGS defined by BFLAGS above
* We can put it into section 2 - sections of elements are ignored


      call dsdef('BOX','S',2,3,'F(2) POSITION FLAGS '//
     >                         'FRAME POINT BFLAGS',iv,ierr)


* Now that we have our BOX we may use it as a template to define
* another one - in section 1 - this defines BOX3 to look like
* BOX.


      call dslike('BOX3','BOX',1,iv,ierr)
```

and the same example in c:

```
int iv,ierr;

/*
 define POINT - REAL, section 1, 3 items (X,Y,Z), extra comments
 for interactive work ('|' separates items if they contain spaces)
 section 0 means that this object is only a template to be used
 in other definitions rather than to store data
 Using a non-zero section is allowed - POINT would then become
 a data set definition as well and one could store data in it.
 Section number is ignored when POINT is used in other definitions.
*/


dsdef("POINT","R",0,3,
  "X|Y|Z|X coordinate|Y coordinate|Z coordinate",&iv,&ierr);
```

```
/*
 Now define FRAME - a structure - an array of 4 points (1 item)
*/

dsdef("FRAME","S",0,1,"C(4) POINT",&iv,&ierr);

/*
 BFLAGS - to keep extra flags - 3 items - 1 integer,1 logical
and one 16 character string - here none of the words
contains spaces, so a space may be used for a separator
(The comments 'Box_colour On_off Box_name'
are optional but recommended)
 Note the second argument describing the data set type
*/

dsdef("BFLAGS","1I 1L 4H",0,3,
"COLOUR ONOFF NAME Box_colour On_off Box_name",&iv,&ierr);

/*
 Now we are ready to define our box - a structure with 3 items
 the first is an array of 2 frames F followed by a point POSITION
 and a structure FLAGS defined by BFLAGS above
 We can put it into section 2 - sections of elements are ignored
*/

dsdef("BOX","S",2,3,
"F(2) POSITION FLAGS FRAME POINT BFLAGS",&iv,&ierr);

/*
 Now that we have our BOX we may use it as a template to define
 another one - in section 1 - this defines BOX3 to look like BOX.
*/

dslike("BOX3","BOX",1,&iv,&ierr);
```

BOX and BOX3 are now defined and we may store data - as many entries as needed in each. They may also be mapped directly into fortran or c structures defined above.

An equivalent definition of BOX could be made without BFLAGS structure by using dspack pre-defined types (.INT4, .LOG4, and .CHAR16 in this case):

```
      call dsdef('BOX','S',2,5,
     >             'F(2) POSITION COLOUR ONOFF NAME'//
     >             'FRAME POINT .INT4 .LOG4 .CHAR16',iv,ierr)
```

which corresponds to a fortran structure like this:

```
structure /BOX/
         record/FRAME/  F(2)
         record/POINT/  POSITION
         Integer*4        COLOUR
         Logical*4        ONOFF
         Character*16     NAME
end structure
```

In the same way POINT could be replaced by .REAL4 with POSITION replaced by POSITION(3). However it should be stressed that changes as these have no practical effect on performance - so the format should be dictated by the convenience of use. (For instance the last dsdef would result in the name of the box to be BOX.NAME rather than BOX.FLAGS.NAME but replacing POINT by .REAL4 would result in BOX.POSITION.Y to become BOX.POSITION(2) which is, if anything, less convenient)

Pre-defined types are just templates defined by dspack and may be treated like any other definitions. There are very few of them in the moment. However - if a programmer wants his box name to have, say 24 characters then a 24 character template can be easily defined:

```
call dsdef('.CHAR24','C*24',0,1,'C Character*24',iv,ierr)
```

.CHAR24 may then be used in any subsequent structure definition.

4 Calling sequences

4.1 General remarks

In order to make calling sequences identical (as much as language permits) from C and FORTRAN an interface library has been written and is included in the dspack library. (as the result +ppu option is needed when compiling fortran programs using dspack on the HP - this allows two routines with the same name - one in fortran and one in C to coexist in the executable program).

Example: The dsdef call above could be written in C like that:

```
if( !dsdef(".CHAR24","C*24",0,1,"C Character*24",&iv,&ierr))
        {
         printf("Error defining .CHAR24: %i\n",ierr);
         exit(-1);
        }
```

All character strings used as arguments in C calls must be NULL terminated, even if they are used to return data.

DSPACK makes no attempt to align data. The mininum data 'chunk' is a 4 byte word. The proper alignment should be ensured when defining objects. DSPACK will always assumme that objects are exacly as they are defined and once the number of 4 byte words per objects entry is calculated it is used when moving data. In particular if an element of a structure is of type character it must have the correct number of bytes independently of NULL termination.

Many routines have the list of arguments starting from

NAME - data set name

IVDSN - data set handle

Unless explicitly stated that 'name' may refer to an 'item' - the name must be a full name of a data set.

IVDSN is a data set handle. DSPACK treats it as follows:

- If IVDSN=0 a search for name is performed and ivdsn is set to correct value. If the search fails IVDSN is set to a negative of the current number of defined data sets.
- If IVDSN>0 and name does not start from '-' - DSPACK checks whether the handle corresponds to the name. If it does not DSPACK acts as if IVDSN=0
- If name starts from '-' DSPACK assummes that ivdsn is correct and does not touch or check it.
- If IVDSN<0 it is checked against the current number of defined data sets. If the number of data sets changed since the last failure for this data set - the search is performed as if IVDSN=0. If no new data sets were defined DSPACK does nothing.

Consequently - a separate variable should be used for ivdsn for each call. It should be preset to 0 before the first call and never touched again. Once found the handle is always valid and allows

DSPACK to find the data quickly. It should be included in a SAVE statement in fortran and declared static in C.

As a convention here all returned arguments are marked with an asterisk (íarg*í) and all arguments which are both input and output are marked with preceeding and following asterisk (í*arg*í)

Fortran and C calls are shown for each routine. Unless explicitly shown otherwise fortran calls are implemented as subroutines. For C calls the return value is usually the same as ierr except if the function returns a pointer.

For succesful return ierr=0.

All arguments containing names are of type character, all others are integer.

4.2 General

4.2.1 DSINIT

```
dsinit(mode,name,lun,msize,msiz2,ierr*)

int dsinit(mode,name,lun,msize,msiz2,ierr)
int   mode;
char  *name;
int   lun;
int   msize;
int   msiz2;
int   *ierr;
```

Initializes dspack. In client-server mode also connects to the server. This is the only call which distinguishes between client-server and local operation.

MODE - operation mode:

- **0** - local - dspack calls are executed by the current process
- **1** - client - dsinit tries to connect to the server
- **2** - server

NAME - server and client names separated by a '/' (for instance 'server-1/patrec')

- **if mode=0** server name is ignored, client name is optional
- **if mode=2** only server name should be specified (and no '/')

LUN - fortran unit number for message output

- **6** - normally standard output
- **0** - no message output

MSIZE - initial size of dspack memory in words (ignored if mode=1)

MSIZ2 - initial size of section 2 (permanent store) in words (ignored if mode=1)

IERR* - error flag

0 - succesful return

4.3 Defining data sets

4.3.1 DSDEF

```
dsdef(name,type,isec,nitem,labels,iv*,ierr*)

int dsdef(name,type,isec,nitem,labels,iv,ierr)
char   *name;
char   *type;
int    isec;
int    nitem;
char   *labels;
int    *iv;
int    *ierr;
```

DSDEF defines a data set. The definition is stored permanently and is used when the operation on data set are performed.

NAME - character - data set name (up to 16 characters).dots and spacess are illegal, names beginning with a dot are considered as internal DSPACK objects

TYPE - character - data set type:

'I' integer

'R' real

'L' logical

'D' double precision

'B' bit string

'C*n' fixed character - n bytes per item (mod(n,4)=0)

'C' character - only single entry of any length

'S' structure

'S' type implies that the data set is defined on a basis of other objects defined already. The 'comment' fields of LABELS array are used to specify names of template objects.

The above type may be followed by 'A' (ie C*24A) - this means that the comment items are present in the LABELS array. If LABELS are passed as a single string 'A' is not necessary.

If the data set is to have items of different types (but no arrays or structures) it may be defined as mixed. In this case TYPE has a form similar to the one used by MZFORM - for instance:

'3I 4R 4H' - 3 integers, 4 reals and one 16 byte character item

'3D 2R 2L' - 3 double, 2 reals and 2 (4-byte) logicals

'1I 2H 3H' - 1 integer, one 8 byte character and one 12 byte character

Separate character items should not be joined together. 'S' and '/' descriptors are forbidden.

ISEC - section number.
by convention:

1 - event section, short term

2 - constant section - long term

....

0 - no section - used to define templates - objects with ISEC=0 cannot hold data - they may be used to define other objects.

NITEM - number of items per entry.

LABELS - character array or character string - item names and additional info.
Two forms are allowed:

character*n LABELS(m) where n may be anything, but only first 20 characters are significant,m must be at least NITEM except in following cases:

1) if 'A' is present in TYPE, m must be at least 2*NITEM. The last NITEM elements contain comments - one for each item.

2) if TYPE='S', m must be at least 2*NITEM. The last NITEM elements contain names of known objects which define items of the data set. Important: First NITEM LABELS must not contain spaces, dots, commas or character '|',

character*m LABELS In this form dsdef does its own parsing of the string LABELS.
Parsing rules:

- Elements are separated by spaces or character '|'. Separators must not be mixed - use either spaces or '|' in the whole string.
- There must be at least NITEM elements.
- If TYPE='S' there must be at least 2*NITEM elements. 'A' in TYPE is not needed in this case and is ignored.

IV* - if succesful dsdef returns the data set handle - in principle this handle could be stored by the program and the data set name could be ignored from then on. However in practice it is easier to use a separate handle for each call in each program unit rather than passing handles between units.

IERR* - dspack error flag - should be 0 when succesful.

4.3.2 DSLIKE

```
dslike(newnam,oldnam,isec,ivdsn*,ierr*)

int dslike(newnam,oldnam,isec,ivdsn,ierr)
char  *newnam, *oldnam;
int   isec, *ivdsn, *ierr;
```

Defines new data set using existing definition.

NEWNAM - new data set name

OLDNAM - existing data set name

ISEC - section number of the new data set

IVDSN* - new data set handle

IERR* - DSPACK error flag

4.4 Storing data

4.4.1 DSBOOK

```
dsbook(name,*ivdsn*,maxent,ierr*)

int dsbook(name, ivdsn, maxent, ierr)
char *name;
int  *ivdsn, maxent, *ierr;
```

Allocates memory for data set. DSBOOK creates a contiguous region of memory to store maxent entries. The entries may be added individually by calling DSADDE.

NAME - data set name

IVDSN - data set handle

MAXENT - Number of entries requested

IERR* - DSPACK error flag

4.4.2 DSADDE

```
dsadde(name,*ivdsn*,nenti,nentot*,ix,ierr*)

int dsadde(name, ivdsn, nenti, nentot, ix, ierr)
char *name;
int *ivdsn, nenti, *nentot, *ix, *ierr;
```

Adds entries to a data set. Should be used after memory was allocated with DSBOOK.

NAME - data set name

IVDSN - data set handle

NENTI - Number of entries to add

NENTOT* - Returns the number of entries after this call

IX - data - array containing **nenti** entries

IERR* - DSPACK error flag

4.4.3 IDSADD

```
      integer function idsadd(name,*ivdsn*,ient,
     >               ngot*,isize*,nentot*,ierr)

void *idsadd(name,ivdsn,ient,ngot,isize,nentot,
                                         ierr)
char *name;
int *ivdsn,ient,*ngot,*isize,*nentot,*ierr;
```

Adds entries to a data set. Works like DSADDE but there is no data copying. The routine returns the pointer to the first of the added entries and the user may write data there.

NAME - data set name

IVDSN - data set handle

IENT - number of entries to add

NGOT* - number of entries actually added

ISIZE* - data set size (per entry) in bytes

NENTOT* - new total number of entries

IERR* - DSPACK error flag

4.4.4 DSPUT

```
dsput(name,*ivdsn*,nent,ix,ierr*)

int dsput(name,ivdsn,nent,ix,ierr)
char *name;
int  *ivdsn, nent, *ix, *ierr;
```

Stores data in previously defined data set. The existing data (if any) are destroyed and **nent** becomes the new number of entries.

NAME - data set name

IVDSN - data set handle

NENT - number of entries

IX - data

IERR* - DSPACK error flag

4.4.5 DSREPL

```
dsrepl(name,*ivdsn*,ient,nent,ix,ierr*)

int dsrepl(name,ivdsn,ient,nent,ix,ierr)
char *name;
int  *ivdsn, ient, nent, *ix, *ierr;
```

Replaces selected entries of existing data set.

NAME - data set name

IVDSN - data set handle

IENT - first entry to replace

NENT - number of entries to replace

IX - data

IERR* - DSPACK error flag

4.4.6 DSDROP

```
dsdrop(name,*ivdsn*,ierr*)

int dsdrop(name,ivdsn,ierr)
char *name;
int  *ivdsn, *ierr;
```

Removes stored data set from memory. Only data are removed, not the definition.

NAME - data set name

IVDSN - data set handle

IERR* - DSPACK error flag

4.4.7 DSHIDE

```
dshide(name,*ivdsn*,ldat*,nent*,ierr*)

int dshide(name,ivdsn,ldat,nent,ierr)
char *name;
int  *ivdsn, *ldat, *nent, *ierr;
```

Makes data set 'invisible' to subsequent calls. The data are preserved and may be restored later. This means that subsequent PUTs make alternative versions of the same data set and the programmer may switch between the sets.

NAME - data set name

IVDSN - data set handle

LDAT* - link to hidden data - together with **nent** it is needed to restore the data set

NENT* - number of entries hidden - needed to restore the data set

IERR* - DSPACK error flag

4.4.8 DSREST

```
dsrest(ivdsn,ldat,nent,ierr*)

int dsrest(ivdsn,ldat,nent,ierr)
int  *ivdsn, *ldat, *nent, *ierr;
```

Restores data hidden with DSHIDE. The hidden data will replace the current data - therefore if DSHIDE was not called for the current data set - it will be lost.

IVDSN - data set handle - as returned by DSHIDE

LDAT - link to hidden data - as returned by DSHIDE

NENT - number of entries hidden - as returned by DSHIDE

IERR* - DSPACK error flag

4.4.9 DSCLR

```
dsclr(name,*ivdsn*,ierr)

int dsclr(name,ivdsn,ierr)
char *name;
int  *ivdsn, *ierr;
```

Clears the contents of the data set to 0 and resets the existing number of entries to 0. After this call DSADDE may be called again to fill the data set with data.

NAME - data set name

IVDSN - data set handle

IERR* - DSPACK error flag

4.4.10 DSCUTL

```
dscutl(ivdsn,nent,ierr*)

int dscutl(ivdsn,nent,ierr)
int  *ivdsn, nent, *ierr;
```

Changes the number of entries of a data set.

IVDSN - data set handle

NENT - new number of entries

IERR* - DSPACK error flag

4.4.11 DSSEL

```
dssel(name,*ivdsn*,maxent,nent*,flist,ierr*)

int dssel(name,ivdsn,maxent,nent,flist,ierr)
char   *name;
int    *ivdsn, maxent, *nent, *flist, *ierr;
```

Removes unwanted entries from the data set. This routine actually moves data and should be used only if it is important that the valid entries are contiguous (for instance before writing data to tape or disk)

NAME - data set name

IVDSN - data set handle

MAXENT - maximum number of entries after selection

NENT* - number of entries after selection

FLIST - logical array of length maxent containing .TRUE. for each entry to be selected, .FALSE. for other entries. (integer (32bit) array may be used instead with 1 and 0 replacing .TRUE. and .FALSE. respectively)

IERR* - DSPACK error flag

4.5 Retrieving data

4.5.1 DSNENT

```
dsnent(name,*ivdsn*,nent*,ierr*)

int dsnent(name,ivdsn,nent,ierr)
char   *name;
int    *ivdsn, *nent, *ierr;
```

Returns present number of entries.

NAME - data set name

IVDSN - data set handle

NENT* - data set handle

IERR* - DSPACK error flag

4.5.2 DSGET

```
dsget(name,*ivdsn*,maxent,nent*,ix*,ierr*)

int dsget(name,ivdsn,maxent,nent,ix,ierr)
char *name;
int  *ivdsn, maxent, *nent, *ix, *ierr;
```

Returns up to maxent entries from the data set. It should be used to retrieve a complete data set - the maxent is only meant as a protection.

NAME - data set name

IVDSN - data set handle

MAXENT - Number of entries requested

NENT* - Returns the number of entries returned

IX* - Array to accept data

IERR* - DSPACK error flag

4.5.3 DSGETS

```
dsgets(name,*ivdsn*,ient,maxent,nent*,ix*,ierr*)

int dsgets(name,ivdsn,ient,maxent,nent,ix,ierr)
char *name;
int  *ivdsn, ient, maxent, *nent, *ix, *ierr;
```

Gets selected entries from a data set. It works like DSGET except that **ient** is the number of the first entry to be returned.

NAME - data set name

IVDSN - data set handle

IENT - first entry to be returned

MAXENT - Number of entries requested

NENT* - Returns the number of entries returned

IX* - Array to accept data

IERR* - DSPACK error flag

4.5.4 IDSGET

```
      integer function idsget(name,*iv*,ient,
     > isize*,ngot*,mxent,ncopy,idata,ierr*)

void *idsget(name, iv, ient, isize, ngot,
             mxent, ncopy, idata, ierr)
      char *name;
      int *iv, ient, *isize, *ngot,
          mxent, *ncopy, *idata, *ierr;
```

Generic GET function.
IDSGET may perform following tasks

- Copy complete data sets into specified address
- Copy selected entries of a data set
- Copy selected words of each entry into a vector
- Copy selected subsets each entry into an array
- Return a pointer to a data set or to any subset

NAME - data set name

IV - descriptor. It helps idsget to avoid multiple searches or name parsing. It must be set to 0 before the first call and never modified by the user.

IENT - entry number (ignored if specified in name)

ISIZE* - ROOT data set size in bytes (per entry)

NGOT* - Number of available entries

MXENT - Number of entries to copy (if 0 only the pointer s returned)

NCOPY* - Number of entries copied

IDATA* - Copied data

IERR* - DSPACK error flag

Example Fortran:

```
      subroutine gt_show1
*
*   ******************************************************************
*   *                                                                *
*$$ *   subroutine gt_show1  demonstration of idsget function        *
*   *                        with data copying                       *
*   *                                                                *
*   *   Ryszard Zybert    Dec  2 12:16:22 1993                       *
*   ******************************************************************
*
      structure /VECT3/
              real  X
              real  Y
              real  Z
      end structure
      structure /POINT/
              real  X
              real  Y
              real  Z
      end structure
      structure /TRACKS/
              real  Q
              record/VECT3/  P
              record/POINT/  VERTEX
              integer  PID
              integer  N_HITS
              integer  FIRST_HIT
      end structure
      record/point/ vl(100)
      record/tracks/ tl(50)
      real zl(1000)
      save
```

```
      data iv1,iv2,iv3/3*0/

      call idsget('TRACKS',iv1,1,is,ngot,50,nc,tl,ierr)
      write(*,*)'Got ', nc, ' tracks'
      write(*,*)(tl(i).vertex.z,i=1,nc)

      call idsget('TRACKS.VERTEX',iv2,1,is,ngot,100,nc,vl,ierr)
      write(*,*)'Got ', nc, ' vertices'
      write(*,*)(vl(i).z,i=1,nc)

      call idsget('TRACKS.VERTEX.Z',iv3,1,is,ngot,1000,nc,zl,ierr)
      write(*,*)'Got ', nc, ' numbers'
      write(*,*)(zl(i),i=1,nc)

      end
      subroutine gt_show2
*
*   ******************************************************************
*   *                                                                *
*$$ *   subroutine gt_show2  demonstration of idsget function          *
*   *                        with pointers                           *
*   *                                                                *
*   *   Ryszard Zybert    Dec  2 12:16:22 1993                       *
*   ******************************************************************
*
      structure /VECT3/
              real  X
              real  Y
              real  Z
      end structure
      structure /POINT/
              real  X
              real  Y
              real  Z
      end structure
      structure /TRACKS/
              real  Q
              record/VECT3/  P
              record/POINT/  VERTEX
              integer  PID
              integer  N_HITS
              integer  FIRST_HIT
      end structure
      record/point/ v
      pointer (p_v,v)
      record/tracks/ t(1)
      pointer (p_t,t)
      real z
      pointer (p_z,z)
      save
      data iv1,iv2,iv3/3*0/

      p_t = idsget('TRACKS',iv1,1,is,ngot,0,0,0,ierr)
```

```
      write(*,*)'Got ', ngot, ' tracks'
      write(*,*)(t(i).vertex.z,i=1,ngot)

      p_v = idsget('TRACKS.VERTEX',iv2,1,is,ngot,0,0,0,ierr)
      write(*,*)'Got ', ngot, ' vertices'
      do i=1,ngot
        write(*,*)v.z
        p_v = p_v + is
      enddo

      p_z = idsget('TRACKS.VERTEX.Z',iv3,1,is,ngot,0,0,0,ierr)
      write(*,*)'Got ', ngot, ' numbers'
      do i=1,ngot
        write(*,*)z
        p_z = p_z + is
      enddo

      end
```

c:

```
int gt_show()
/*
*   ********************************************************************
*   *                                                                  *
*$$ *   int gt_show  demonstration of idsget function                  *
*   *                                                                  *
*   *   Ryszard Zybert    Dec  2 12:12:42 1993                         *
*   ********************************************************************
*/
{
typedef struct VECT3
{
       float X;
       float Y;
       float Z;
};
typedef struct POINT
{
       float X;
       float Y;
       float Z;
};
typedef struct TRACKS
{
       float Q;
       struct VECT3 P;
       struct POINT VERTEX;
       int PID;
       int N_HITS;
       int FIRST_HIT;
};
```

```
struct POINT vl[100];
struct TRACKS tl[50];
float zl[1000];

struct POINT *v;
struct TRACKS *t;
float *z;

static int iv1=0, iv2=0, iv3=0;
int ngot,nc,ierr,is,i,nd;

/* using pointers - no copying */

t = (struct TRACKS *)idsget("TRACKS",&iv1,1,&is,&ngot,0,0,0,&ierr);
printf("Got %i tracks\n",ngot);
for( i=0; i<ngot; i++ )
  printf("%f\n",(t++)->VERTEX.Z);

idsget("TRACKS.VERTEX",&iv2,1,&is,&ngot,0,0,0,&ierr);
printf("Got %i vertices\n",ngot);
for( i=1; i<=ngot; i++)
  {
  v=(struct POINT *)idsget("TRACKS.VERTEX",&iv2,i,&is,&nd,0,0,0,&ierr);
  printf("%f\n",v->Z);
  }

printf("\n");
nd = 2;
i = 0;
while( nd>1)
  {
  z=(float *)idsget("TRACKS.VERTEX.Z",&iv3,++i,&is,&nd,0,0,0,&ierr);
  printf("%i %f\n",i,*z);
  }

/* copying data */

idsget("TRACKS",&iv1,1,&is,&ngot,50,&nc,tl,&ierr);
printf("Got %i tracks\n",nc);
for( i=0; i<nc; i++ )
  printf("%f\n",tl[i].VERTEX.Z);

idsget("TRACKS.VERTEX",&iv2,1,&is,&ngot,100,&nc,vl,&ierr);
printf("Got %i vertices\n",nc);
for( i=0; i<nc; i++ )
  printf("%f\n",vl[i].Z);

idsget("TRACKS.VERTEX.Z",&iv3,1,&is,&ngot,1000,&nc,zl,&ierr);
printf("Got %i numbers\n",nc);
for( i=0; i<nc; i++ )
  printf("%f\n",zl[i]);

return 0;
```

}

4.6 Retrieving data set information

4.6.1 DSIVID

```
dsivid(name,label,ivdsn*,item*,ierr*)

int dsivid(name,label,ivdsn,item,ierr)
char *name, *label;
int  *ivdsn, *item, *ierr;
```

Returns data set handle and item number.

NAME - data set name

LABEL - item name

IVDSN - data set handle

ITEM* - item number

IERR* - DSPACK error flag

4.6.2 DSL1L2

```
dsl1l2(ivdsn,item,name*,label*,ierr*)

int dsl1l2(ivdsn,item,name,label,ierr)
char *name, *label;
int  *ivdsn, *item, *ierr;
```

Returns data set name and item name.

IVDSN - data set handle

ITEM - item number

NAME* - data set name

LABEL* - item name

IERR* - DSPACK error flag

4.6.3 DSHDRF

```
dshdrf(name,*ivdsn*,*icnt*,lun,ierr*)

int dshdrf(name,ivdsn,icnt,lun,ierr)
char   *name;
int    *ivdsn, *icnt, lun, *ierr;
```

Prints f77 header file entry for data set name. All structures needed for the data set definition are printed.

NAME - data set name

IVDSN - data set handle

ICNT - 'DONE' counter. It allows the program to avoid repeating the definitions if they are referenced more than once. If the output is to be written into a header file, icnt should be set to 0 on the first call for this file and not modified by the calling program until the last call for this file.

LUN - Fortran logical unit for output

IERR* - DSPACK error flag

4.6.4 DSSHDC

```
dsshdc(name,*ivdsn*,*icnt*,lun,ierr*)

int dsshdc(name,ivdsn,icnt,lun,ierr)
char   *name;
int    *ivdsn, *icnt, lun, *ierr;
```

Prints c header file entry for data set name. All structures needed for the data set definition are printed.

NAME - data set name

IVDSN - data set handle

ICNT - 'DONE' counter. It allows the program to avoid repeatingthe definitions if they are referenced more than once. If the output is to be written into a header file, icnt should be set to 0 on the first call for this file and not modified by the calling program until the last call for this file.

LUN - Fortran logical unit for output

IERR* - DSPACK error flag

4.6.5 DSINPH

```
dsinph(ioid,lun)

int dsinph(ioid,lun)
int   ioid, lun;
```

Produces f77 header file which defines all data sets found on input ID.

IOID - input ID

LUN - Fortran unit number for output

4.6.6 DSINPC

```
dsinpc(ioid,lun)

int dsinpc(ioid,lun)
int   ioid, lun;
```

Produces c header file which defines all data sets found on input ID.

IOID - input ID

LUN - Fortran unit number for output

4.6.7 DSALLH

```
dsallh(lun)

int dsallh(lun)
int   lun;
```

Produces f77 header file for all defined data.

LUN - Fortran unit number for output

4.6.8 DSALLC

```
dsallc(lun)

int dsallc(lun)
int   lun;
```

Produces c header file for all defined data.

LUN - Fortran unit number for output

4.6.9 DSNIVR

```
dsnivr(name,*ivdsn*,ierr*)

int dsnivr(name,ivdsn,ierr)
char   *name;
int    *ivdsn, *ierr;
```

Returns data set handle.

NAME - data set name

IVDSN - data set handle

IERR* - DSPACK error flag

4.6.10 DSNCAT

```
dsncat(ivdsn,typ*,isec*,nitem*,ierr*)

int dsncat(ivdsn,typ,isec,nitem,ierr)
char   *typ;
int    *ivdsn, *isec, *nitem, *ierr
```

Returns data set definition info.

IVDSN - data set handle

TYP* - data set type (as in DSDEF)

ISEC* - section number

NITEM* - number of items

IERR* - DSPACK error flag

4.6.11 DSSHOW

```
dsshow(name,ient,lun,ierr*)

int dsshow(name,ient,lun,ierr)
char   *name;
int    ient, lun, *ierr;
```

Displays formatted output with contents of the data set. Either selected entries or the whole data set may be displayed. NAME may describe the data set or a subset. Also entry numbers may be specified in name (see **Access to data by name**).

NAME - data set name or subset name

IENT - entry number. 0 means 'all entries'

LUN - Fortran unit number for output

IERR* - DSPACK error flag

4.6.12 DS_EXAMINE

```
ds_examine(text)

int ds_examine(text)
char   *text;
```

Interactive examination of data. If text is blank the routine enters interactive mode (on standard input and output) to allow the user to interogate the contents of memory - type '?' to see the possible commands. If text is non-blank it is treated as if it was typed from the keyboard, it is executed and the routine returns.

To exit from interactive mode type $< EOF >$ (usually CNTR D).

TEXT - command text. If blank, interactive mode is entered.

4.6.13 DSRIDF

```
dsridf(name,*ivdsn*,typ*,nitem*,lbl2*,ierr*)

int dsridf(name,ivdsn,typ,nitem,lbl2,ierr)
char   *name, *lbl2;
int    *ivdsn, *nitem, *ierr;
```

Returns data set or subset definition

NAME - data set name or full item name

IVDSN - data set handle

TYP* - data type - like in DSDEF

NITEM* - number of items

LBL2* - character*(*) lbl2(*) array with item names

IERR* - DSPACK error flag

4.7 Input/Output

4.7.1 Comments

In the client-server mode all I/O is performed by the server. The calls described here are used by the client to issue instructions to the server. In local mode the calls are performed locally. This is true for many DSPACK calls, however - it is important to realize the implications for the I/O operations - for instance - if the client 'opens' a file and then exits - the file stays open and positioned where the client left it. Another client may 'read' from this file (or 'write' to this file). This is because 'open' 'read' 'write' etc. are in fact commands to the server to do things.

4.7.2 FSOPIN

```
fsopin(ioid,file,chopt,ierr*)

int fsopin(ioid ,file, chopt, ierr)
      int ioid;
      char *file, *chopt;
      int *ierr;
```

Opens DSPACK input file or tape

IOID - input ID (between 0 and 9)

FILE - full path of the file or tape unit

CHOPT - options

- **'T'** for tape
- **'D'** for disk (default)

IERR* - DSPACK error flag

4.7.3 FSOPOU

```
fsopou(ioid,file,chopt,ierr*)

int fsopou(ioid ,file, chopt, ierr)
      int ioid;
      char *file, *chopt;
      int *ierr;
```

Opens DSPACK output file or tape

IOID - output ID (between 0 and 9) - independent of input ID

FILE - full path of the file or tape unit

CHOPT - options

'T' for tape

'D' for disk (default)

IERR* - DSPACK error flag

4.7.4 FDSIN

```
fdsin(ioid,ktyp*,ierr*)

int fdsin(ioid, ktyp, ierr)
int ioid, *ktyp, *ierr;
```

Reads one record

IOID - input ID (must be previously opened)

KTYP* - record type read

0 definition block

1 data for section 1 (variables)

2 data for section 2 (constants)

IERR* - DSPACK error flag

4.7.5 FSMARK

```
fsmark(ioid,name,ntot*,ierr*)

int fsmark(ioid, name, ntot, ierr)
int ioid, *ntot, *ierr;
char *name;
```

Marks data set for output. After all required data sets are marked FSOUTD should be called to output the definition block. Following FSOUTV calls will write out all marked data sets for given output ID if they are stored in the section to which FSOUTV call refers. This way all sets are marked, independently of section but contants (section 2) and variables (section 1) are written out as separate records.

IOID - output ID

NAME - data set name

NTOT* - number of data sets marked for this **ioid**

IERR* - DSPACK error flag

4.7.6 FSOUTD

```
fsoutd(ioid,ierr*)

int fsoutd(ioid, ierr)
int ioid, *ierr;
```

Writes definitions of all data sets marked for this **ioid**. It must be called after all FSMARK calls were made and before any FSOUTV call.

IOID - output ID

IERR* - DSPACK error flag

4.7.7 FSOUTV

```
fsoutv(ioid,isect,nout*,ierr*)

int fsoutv(ioid, isect, nout, ierr)
int ioid, isect, *nout, *ierr;
```

Writes all data sets marked for this ID if they are stored in the specified section.

IOID - output ID

ISECT - section number

NOUT* - number of data sets written

IERR* - DSPACK error flag

5 Environment

In 'local' mode DSPACK does not depend on any environment variables (at present). In client-server mode there is a single variable DSPACK_FILES which points to the location of FIFOs used for communication, It may depend on the local installation where the FIFOs are placed. Usually DSPACK_FILES points to the top DSPACK directory, It may be convenient to set DSPACK_HOME to the top DSPACK directory - for easy access to binaries, libraries and examples. In this case DSPACK_FILES=$DSPACK_HOME would be a normal setting.

5.1 Initialization file

During initialization DSPACK tries to locate the initialization file .dspackrc. The search order is as follows:

1 - Find a file pointed to by $DSPACKRC

2 - Find ./.dspackrc

3 - Find ~/.dspackrc

The format of the .dspackrc file is described by comments inside file $DSPACK_HOME/.dspackrc, the basic rules are:

- Each item has a form T1.T2.T3 : value comment
  Where:

  T1 - is 'dspack' or '*'

  T2 - is the client or program name as given to dsinit - or '*'

  T3 - option name

  comment - may start anywhere after a valid value for the option

  For instance a line

  ```
  dspack.mzt.case : off      No case converion
  ```

  means that keyboard entries are treated 'as typed' and not converted to upper cases - this line is only seen by the program mzt and is ignored by all other programs.

- dspack.<program>.end: causes <program> to skip the rest of the file

- dspack.<program>.setenv: <name> = <value> causes setting the envirnomental variable <name>. This variable may be later used by the program. <value> is expanded if it contains existing environment variables. The typical example may be:

  ```
  dspack.*.setenv :    DSPACK_FILES = $DSPACK_HOME
  ```

- dspack.<program>.message : <message text> causes <message text> to be send to standard output. (This is disabled if LUN=0 in dsinit call).

- Spaces are usually irrelevant (except for message). Case is irrelevant on the left side of the colon, the right had side may or may not be case sensitive depending on the option.

- Lines which were not recognized as known options are passed to user's routine ds_user_widget(name,value) - where name and value are character variables containing the left and the right side of the colon. A dummy ds_user_widget is in the dspack library. Lines without a colon are ignored.

6 Portability

DSPACK was written to run under UNIX system. Although the system dependent elements were limited to minimum and isolated - much of the program is written with some use of f77 extensions of Fortran 77 compiler.

Most of the extensions are compatible with the VMS FORTRAN, however - some modifications to the code will be necessary. The matter is complicated more as some of the DSPACK routines return pointers to data - on VMS they may still return a pointer, but the user's fortran program cannot use them directly as it can with f77. This problem does not occur with the user's C code.

At present DSPACK has been tested on HP and SunOS. The plan is to make it available on as many UNIX platforms as necessary. A single user version is running on a Macintosh - compiled with the ABSOFT Fortran ocmpiler under MPW. With time, the parts of the code which rely on f77 extentions will be probably replaced by C code.

Wherever necessary the different versions of the code are kept in the source and the C preprocessor is used to select the code for the host machine.

DSPACK relies to a minimal extent on the CERN program library. However as this is only required for a few simple routines they could - if necessary be replaced locally.

7 Installation

DSPACK distribution set is kept in **/users/zybert/dspack/dspack.tar.Z** on **hpna49-1.cern.ch**. It should be copied to a directory which is to be DSPACK home directory. Installation steps:

1) - **setenv DSPACK_HOME** to where the DSPACK should be

2) - **cd $DSPACK_HOME**

3) - **uncompress dspack.tar.Z**

4) - **tar xvf dspack.tar**

5) - edit **src/local.mk** to conform to local settings (there are **hp.mk** and **sun.mk** files there as examples)

6) - **make install**

This should compile DSPACK and examples make a simple server **dsserver** which is used by all examples. The server should be run in background and once it is running you can try the examples. **make install** also makes a directory **/tmp/dspack/fifo** which is used for interprocess communication.

8 Glossary

Throughout this guide several terms appear repeatedly. It may be useful to explain them here.

Data set DSPACK allows you to define an object in the way similar to C or Fortran structure declaration. **Data set** refers to all data stored following this definition. Each data set requires a separate, unique definition. The definition implies that an array of structures may be saved this way and in this case **Data set** refers to the whole array.

Data set entry If the data set is an array - the **entry** refers to one element of the array.

Data set size The size in bytes of one entry of the data set.

Data set item If the data set is a structure, then each element of this structure is called **item** . The **item** may be itself a structure. The **item** has a name (up to 16 characters) which must be unique within the data set.

Data set name The data set may be referred to by a unique name which is set when the set is defined. The name may be up to 16 characters long and must not contain spaces or dots (except that internal DSPACK objects have names starting with a dot). The name must not start from a '-' sign. If a '-' sign appears as a first character of the name when the data set is referred to DSPACK assumes that the name should not be used and the data set should be found directly using the handle or call descriptor.

Data set handle Each defined data set has a reserved location in the DSPACK internal tables where pointers to the data set definition, to the data and some other information about the data set are stored. The **handle** is the link to this location. In most DSPACK calls the **handle** is used together with the data set name. The **handle** of the data set never changes once the data set is defined. The user presets the variable to be used to store the handle to zero before the first call referring to the given data set. DSPACK finds the data set by name and returns correct **handle** . In subsequent calls the **handle** is only checked against the name.

DSPACK section DSPACK stores data in two separate sections - section 1 is meant for short term data and would usually be cleared before each event is processed, section 2 is permanent and should be used to store constants. More sections may be defined if necessary. DSPACK treats all sections in the same way with one exception: when reading DSPACK file or tape data are read-in into section 1 and, if necessary copied to the appropriate section. This allows the existing data sets in sections other than section 1 to be overwritten by the data from input. In the output data from different sections are stored in separate records - generated by separate calls of the output routine. Usually the section 2 would be written out twice, at the beginning and at the end of the output file, section 1 would be written out after each event.

ROOT data set If the name used in the DSPACK call refers to an item of a data set rather than the whole data set, the **ROOT data set** is the data set referred to by the first element of this name.(for instance 'TRACKS' is the **ROOT data set** in all calls in the examples of using IDSGET routine)

Contents