Part I

DSPACK – Overview

Chapter 1: Introduction

A large software package as that needed to process data from a large HEP 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.

A common temptation when writing a package like this is to try to solve all possible problems at once. DSPACK follows a different philosophy. It applies solutions which should provide optimum performance for the HEP applications and provides 'hooks' which should be general enough to satisfy other needs.

DSPACK has been written with UNIX operating system in mind. However - with some modifications it should run on come other systems.

Chapter 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.

- Object Description Files may be used to define objects or to initialize them. The format of Object Description Files allows for comments and should be useful for documentation purposes. (see 7)

- Object definitions are expandable - the user is free to add data to the object definition. (see 10)

- 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 preserves the DSPACK's calling sequences.

- Concepts of Object Oriented Programming are implemented. Data set (or 'Object') can be assigned to a CLASS. The mechanism of passing messages to objects is available. The CLASS system is intended to enable the user to extend DSPACK in order to handle special objects in special way. It may be used to build interfaces to other packages or to provide functionality which does not exist in DSPACK.(see 6)

Chapter 3: Glossary

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

3.0.0.1 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.

3.0.0.2 Data set entry

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

3.0.0.3 Data set size

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

3.0.0.4 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.

3.0.0.5 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.

3.0.0.6 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.

3.0.0.7 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.

3.0.0.8 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)

Chapter 4: 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.

4.1 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.

4.2 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:

4.2.0.9 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.

4.2.0.10 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.

4.2.0.11 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 or Fortran would be something like that:

Example of C and Fortran structures

```
 In C
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;
```

Or in Fortran

```
        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).

4.2.0.12 Structure definitions 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.

CERN Program Library Long Writeups XXXX

DSPACK

Object-Oriented Data Manager

Tutorial

Version 1.13

Application Software Group

Computing and Networks Division

CERN Geneva, Switzerland

CERN Program Library Office
CERN-CN Division
CH-1211 Geneva 23
Switzerland
Tel. +41 22 767 4951
Fax. +41 22 767 7155
Bitnet: CERNLIB@CERNVM
DECnet: VXCERN::CERNLIB (node 22.190)
Internet: CERNLIB@CERNVM.CERN.CH

Contact Person: Ryszard Zybert (R.Zybert@bham.ac.uk)

Contact Person: Predrag Buncic (Predrag.Buncic@cern.ch)

Preliminary remarks

This **Complete Reference** of the DSPACK system, consists of four parts:

1 An **overview** of the system.
2 A **step by step tutorial introduction** to the system.
3 A **reference guide**, describing each command in detail.
4 An **installation and management guide**.

The DSPACK system is implemented on Hewlett-Packard, IBM RS6000, and SUN. Work on porting it to other machines is in progress.

This document has been produced using LaTeX [?] with the cernman style option, developed at CERN.

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.

Table of Contents

Part II

DSPACK – Tutorial

4.3 Tutorial

DSPACK is a data manager. It allows the user to store data in a form which makes them accessible by other programs. The objects (*Data Sets*) correspond to arrays of structures in C or Fortran and the package allows the user to move data easily between local and DSPACK memory.

If the user's program uses pointers, then the distinction between *local* data and *common* data is almost non-existent.

4.3.1 Defining objects

Structures are used widely in C. Most modern Fortran compilers also provide structures, records and pointers, although there are some (platform dependent) limitations. Throughout this manual we use examples tested with f77 compiler on SunOS and HPUX, although they should also work on other platforms.

Example of user's structure in Fortran

```
      structure/track_t/
              real x_start
              real y_start
              real z_start
              real px
              real py
              real pz
              real dx
              real dy
              real dz
              real dpx
              real dpy
              real dpz
              real charge
              real chisq
              integer n_hits
              integer flag1
              integer flag2
              integer point_p              ! pointer to first hit
              integer vertex               ! pointer to vertex structure
              integer nextvt_p             ! pointer to next vertex track
      end structure
```

Similarly one can define a track in C:

Example of user's structure in c

```
typedef struct track_t {
              float x_start;
              float y_start;
              float z_start;
              float px;
              float py;
              float pz;
              float dx;
              float dy;
```

```
            float dz;
            float dpx;
            float dpy;
            float dpz;
            float charge;
            float chisq;
            int n_hits;
            int flag1;
            int flag2;
            point_t *point_p;                /* pointer to first hit */
            vertex_t *vertex_p;              /* pointer to vertex structure */
            track_t *nextvt_p;               /* pointer to next vertex track */
 };
```

The definitions above describe the same object and it is easy to map between them by passing a pointer between programs. The problem arises when we need to write data into a disk file or tape or when we want to access the data interactively.

In DSPACK the track structure may be defined in a similar way:

Structure definition in DSPACK

```
DEFINE track_t S 1                 ! define structure track_t, section 1
              .REAL4 x_start
              .REAL4 y_start
              .REAL4 z_start
              .REAL4 px
              .REAL4 py
              .REAL4 pz
              .REAL4 dx
              .REAL4 dy
              .REAL4 dz
              .REAL4 dpx
              .REAL4 dpy
              .REAL4 dpz
              .REAL4 charge
              .REAL4 chisq
              .INT4  n_hits
              .INT4 flag1
              .INT4 flag2
               point_t *point_p                ! pointer to first hit
               vertex_t *vertex_p              ! pointer to vertex structure
               track_t *nextvt_p               ! pointer to next vertex track
 END DEFINE
```

Some items in this example require explanation.

- The first line in the definition contains an 'S' - this means that the object is a structure - it may contain sub-objects (*items*) of different types.
- '1' in the same line is the section number. It means the the object is a variable (rather than a constant - section 2, or a template - section 0)

- .REAL4 and .INT4 are names of pre-defined DSPACK templates. Elements of a structure in DSPACK must be either structures or pointers to structures. DSPACK defines some simple templates, containing just single numbers just for this purpose. The user can define new objects or templates and use them in other definitions. (templates are kept in 'section' 0) For instance - .INT4 is defined in the following way:

```
DEFINE .INT4 I 0              ! A template of type INTEGER
     "Integer number" I       ! Comment instead of type
END DEFINE
```

- The definition assumes that **point_t** and **vertex_t** are known DSPACK objects. In contrast to C DSPACK does not allow pointers to templates.

- The ∗ in the pointer name is obligatory and it is a part of the name.

Just like in C or Fortran we can simplify the definition by grouping elements:

Structure definition in DSPACK

```
DEFINE V3 R 0                      ! A template for three real numbers
    "X component" x
    "Y component" y
    "Z component" z
END DEFINE

DEFINE track_t S 1                 ! define structure track_t, section 1
                V3 start           ! start position
                V3 p               ! momentum
                V3 d               ! position errors
                V3 dp              ! momentum errors
               .REAL4 charge
               .REAL4 chisq
               .INT4  n_hits
               .INT4  flags(2)
                point_t *point_p                ! pointer to first hit
                vertex_t *vertex_p              ! pointer to vertex structure
                track_t *nextvt_p               ! pointer to next vertex track
END DEFINE
```

To see how this works you need a little program which will read the definitions and interpret them.

example_1.f - test program for example_1

```
      program example_1
*
*   *****************************************************************
*   *                                                               *
*$$ *   program example_1  DSPACK tutorial example_1                *
*   *                                                               *
*   *   Ryszard Zybert    Jul  1 12:58:10 1994                      *
*   *****************************************************************
*

* Initialise DSPACK, local mode, 2MB memory

      call dsinit(0,´example_1´,6,500000,50000,ierr)

* Call ds_examine and do the rest interactively

      call ds_examine(´ ´)

      end
```

This program and the definition file are kept in $DSPACK_HOME/doc/tutorial. The example session below demonstrates how it may be used.

Example session with example_1

```
tmp> mkdir dspack_test
tmp> setenv DSPACK_HOME /na49/dspack/new

tmp> cd $DSPACK_HOME/doc
/na49/dspack/new/doc

doc> tar cf - tutorial | (cd /data/na49/tmp/dspack_test;  tar xf - )

doc> cd /data/na49/tmp/dspack_test/tutorial/examples
/data/na49/tmp/dspack_test/tutorial/examples

examples> make example_1
f77 -c -xl -g -DSUN -I/na49/dspack/new/inc -o example_1.o example_1.f
example_1.f:
 MAIN example_1:
f77 -o example_1 -xl -g -DSUN -I/na49/dspack/new/inc example_1.o
/na49/dspack/new/lib/libdspack.a /na49/dspack/new/lib/zemu.a
/na49/dspack/new/lib/libdspack.a `cernlib genlib packlib kernlib`

examples> example_1
DSPACK 1.129, 30 Jun 1994
dsdb>  errors    ! enable all error messages and warnings
Errors enabled
dsdb>  echo      ! enable echoing of *.d files to the screen
 Echo on
dsdb>  read dspack_tut_1.d
*
*   *****************************************************************
*   *                                                               *
```

```
*$$ *   dspack_tut_1.d  DSPACK tutorial - example 1                  *
*   *                                                                *
*   ******************************************************************
*
DEFINE V3 R 0                       ! A template for three real numbers
    "X component" x
    "Y component" y
    "Z component" z
END DEFINE

DEFINE track_t S 1                  ! define structure track_t, section 1
                V3 start            ! start position
                V3 p                ! momentum
                V3 d                ! position errors
                V3 dp               ! momentum errors
               .REAL4 charge
               .REAL4 chisq
               .INT4  n_hits
               .INT4  flags(2)
                point_t *point_p                 ! pointer to first hit
                vertex_t *vertex_p               ! pointer to vertex structure
                track_t *nextvt_p                ! pointer to next vertex track
END DEFINE
DS_FIXPNT Warning: Void pointer: point_t
DS_FIXPNT Warning: From: track_t
DS_FIXPNT Warning: Void pointer: vertex_t
DS_FIXPNT Warning: From: track_t

dsdb>  ls -l
Name             Type        Id     Sect    Size    Items   Nent    Bytes
V3                R          17       0       3       3       0         0
track_t           S          18       1      20      11       0         0

dsdb>  c                             ! use C mode to display definitions
dsdb>  track_t
 Definition of data set: track_t            (trac) ID:    18
 Type:                   Structure
 Section:                    1
 Machine words:             20
 Items:                     11
   1  start             V3
   2  p                 V3
   3  d                 V3
   4  dp                V3
   5  charge            .REAL4
   6  chisq             .REAL4
   7  n_hits            .INT4
   8  flags(2)          .INT4
   9  *point_p          point_t
  10  *vertex_p         vertex_t
  11  *nextvt_p         track_t

typedef struct V3
{
                float x;
                float y;
                float z;
```

```
};
typedef struct track_t
{
                struct V3 start;
                struct V3 p;
                struct V3 d;
                struct V3 dp;
                float charge;
                float chisq;
                int n_hits;
                int flags[2];
                struct point_t *point_p;
                struct vertex_t *vertex_p;
                struct track_t *nextvt_p;
};
dsdb>  quit

examples>
```

Reading the file produced warnings because two of the pointers defined in the structure refer to unknown objects. We can define the needed objects later, otherwise the pointers will always be set to 0.

4.3.2 .dspackrc file

$DSPACK_HOME/doc/tutorial contains a file called **dspackrc_1**. (by now it should be in the dspack_test/tutorial). This file may be used as a start-up file for any DSPACK program.

$DSPACK_HOME/doc/tutorial/dspackrc_1

```
dspack.*.print            : dspackrc_1 file in $DSPACK_HOME/doc/tutorial

dspack.example_1.print    : *********************
dspack.example_1.print    : * This is Example 1 *
dspack.example_1.print    : *********************
dspack.example_1.echo     : on
dspack.example_1.errors   : all
dspack.example_1.read     : dspack_tut_1.d
dspack.example_1.end      :
```

This file executes the initial commands which the user had to type in the example above. By default DSPACK looks for a file called *.dspackrc* in the current directory or in the user's home directory but we can change it by setting environment variable $DSPACKRC.

Running Example 1 with dspackrc file

```
examples> setenv DSPACKRC ../dspackrc_1

examples> example_1
 dspackrc_1 file in $DSPACK_HOME/doc/tutorial
 *********************
 * This is Example 1 *
 *********************
```

```
DSPACK 1.129, 30 Jun 1994
*
*   *****************************************************************
*   *                                                               *
*$$ *   dspack_tut_1.d  DSPACK tutorial - example 1                 *
*   *                                                               *
*   *****************************************************************
*
DEFINE V3 R 0                      ! A template for three real numbers
    "X component" x
    "Y component" y
    "Z component" z
END DEFINE

DEFINE track_t S 1                 ! define structure track_t, section 1
                V3 start           ! start position
                V3 p               ! momentum
                V3 d               ! position errors
                V3 dp              ! momentum errors
               .REAL4 charge
               .REAL4 chisq
               .INT4  n_hits
               .INT4  flags(2)
                point_t *point_p               ! pointer to first hit
                vertex_t *vertex_p             ! pointer to vertex structure
                track_t *nextvt_p              ! pointer to next vertex track
END DEFINE
DS_FIXPNT Warning: Void pointer: point_t
DS_FIXPNT Warning: From: track_t
DS_FIXPNT Warning: Void pointer: vertex_t
DS_FIXPNT Warning: From: track_t

dsdb>  quit

examples>
```

4.3.3 Command line arguments

Envinronment variables may be set on the command line using the following syntax:

variable=value

There are several mechanisms in DSPACK to use the environment variables. Here is one example:

dspackrc_2 file

```
dspack.*.print           : dspackrc_2 file in $DSPACK_HOME/doc/tutorial
dspack.*.echo            : on
dspack.*.errors          : all

dspack.example_1.print   : ********************************
dspack.example_1.print   : * This is Example 1 (version 2) *
dspack.example_1.print   : ********************************
dspack.example_1.read    : $tut_file
dspack.example_1.end     :
```

```
dspack.example_2.print     : ********************
dspack.example_2.print     : * This is Example 2 *
dspack.example_2.print     : ********************
dspack.example_2.echo      : off
dspack.example_2.read      : $tut_file
dspack.example_2.end       :
```

This file will make DSPACK try to read the file pointed to by the environment variable **tut_file**. This file also demonstrates how different programs may be controlled from the same file by use of program name (example_1, example_2) - the user controls the name by an argument of **dsinit** call.

In the example below the value of **tut_file** is set on the command line.

Command line arguments

```
examples> setenv DSPACKRC ../dspackrc_2

examples> example_1 tut_file=dspack_tut_1.d
 dspackrc_2 file in $DSPACK_HOME/doc/tutorial
 ********************************
 * This is Example 1 (version 2) *
 ********************************
DSPACK 1.130,  5 Jul 1994
*
*   ******************************************************************
*   *                                                                *
*$$ *   dspack_tut_1.d  DSPACK tutorial - example 1                 *
*   *                                                                *
*   ******************************************************************
*
DEFINE V3 R 0                     ! A template for three real numbers
    "X component" x
    "Y component" y
    "Z component" z
END DEFINE

DEFINE track_t S 1                ! define structure track_t, section 1
                V3 start          ! start position
                V3 p              ! momentum
                V3 d              ! position errors
                V3 dp             ! momentum errors
               .REAL4 charge
               .REAL4 chisq
               .INT4  n_hits
               .INT4  flags(2)
                point_t *point_p              ! pointer to first hit
                vertex_t *vertex_p            ! pointer to vertex structure
                track_t *nextvt_p             ! pointer to next vertex track
END DEFINE
DS_FIXPNT Warning: Void pointer: point_t
DS_FIXPNT Warning: From: track_t
DS_FIXPNT Warning: Void pointer: vertex_t
DS_FIXPNT Warning: From: track_t
```

4.3.4 More Object Description Files

The objects misssing in the **track_t** definition are defined in another file: **dspack_tut_2.d**.

dspack_tut_2.d file

```
*
*   ******************************************************************
*   *                                                                *
*$$ *   dspack_tut_2.d  DSPACK tutorial - example 1                  *
*   *                                                                *
*   ******************************************************************
*

DEFINE point_t S 1                ! define structure point_t, section 1
               V3 pos             ! position
               V3 d               ! position errors
              .INT4  flags(2)
               point_t *next      ! pointer to the next hit on track
END DEFINE

DEFINE vertex_t S 1               ! define structure point_t, section 1
               V3 pos             ! position
               V3 d               ! position errors
              .INT4 n_tracks      ! number of tracks
              .INT4  flags(2)
               track_t *tracks    ! pointer to tracks
END DEFINE
```

We could read this file after reading the first one, but it is easier to use **INCLUDE** keyword to read all needed files [1].

dspack_tut_3.d file

```
*
*   ******************************************************************
*   *                                                                *
*$$ *   dspack_tut_3.d  DSPACK tutorial - example 1                  *
*   *                                                                *
*   ******************************************************************
*
INCLUDE dspack_tut_1.d
INCLUDE dspack_tut_2.d
```

INCLUDE can be used at any place in the object description file. It is recursive, up to 10 levels are allowed. All constants defined using **SET** keyword are valid in all included files.

[1] Pointer references need to be resolved when objects are used, however the 'embedded' structure references have to be resolved when the object is defined - therefore file **dspack_tut_1.d** has to be read first in order to define **V3**

The example below reads all the definitions:

Using INCLUDE

```
examples> example_1 tut_file=dspack_tut_3.d
 dspackrc_2 file in $DSPACK_HOME/doc/tutorial
 ********************************
 * This is Example 1 (version 2) *
 ********************************
DSPACK 1.130,  5 Jul 1994
*
*   ******************************************************************
*   *                                                                *
*$$ *   dspack_tut_3.d  DSPACK tutorial - example 1                  *
*   *                                                                *
*   ******************************************************************
*
INCLUDE dspack_tut_1.d
*
*   ******************************************************************
*   *                                                                *
*$$ *   dspack_tut_1.d  DSPACK tutorial - example 1                  *
*   *                                                                *
*   ******************************************************************
*
DEFINE V3 R 0                    ! A template for three real numbers
    "X component" x
    "Y component" y
    "Z component" z
END DEFINE

DEFINE track_t S 1               ! define structure track_t, section 1
                V3 start         ! start position
                V3 p             ! momentum
                V3 d             ! position errors
                V3 dp            ! momentum errors
               .REAL4 charge
               .REAL4 chisq
               .INT4  n_hits
               .INT4  flags(2)
                point_t *point_p              ! pointer to first hit
                vertex_t *vertex_p            ! pointer to vertex structure
                track_t *nextvt_p             ! pointer to next vertex track
END DEFINE
INCLUDE dspack_tut_2.d
*
*   ******************************************************************
*   *                                                                *
*$$ *   dspack_tut_2.d  DSPACK tutorial - example 1                  *
*   *                                                                *
*   ******************************************************************
*

DEFINE point_t S 1               ! define structure point_t, section 1
                V3 pos           ! position
                V3 d             ! position errors
               .INT4  flags(2)
```

```
                point_t *next       ! pointer to the next hit on track
END DEFINE

DEFINE vertex_t S 1                 ! define structure point_t, section 1
                V3 pos              ! position
                V3 d                ! position errors
               .INT4 n_tracks       ! number of tracks
               .INT4  flags(2)
                track_t *tracks     ! pointer to tracks
END DEFINE
dsdb>  c
dsdb>  track_t
 Definition of data set: track_t            (trac) ID:    19
 Type:                  Structure
 Section:                   1
 Machine words:            20
 Items:                    11
   1  start            V3
   2  p                V3
   3  d                V3
   4  dp               V3
   5  charge           .REAL4
   6  chisq            .REAL4
   7  n_hits           .INT4
   8  flags(2)         .INT4
   9  *point_p         point_t
  10  *vertex_p        vertex_t
  11  *nextvt_p        track_t

typedef struct V3
{
                float x;
                float y;
                float z;
};
typedef struct point_t
{
                struct V3 pos;
                struct V3 d;
                int flags[2];
                struct point_t *next;
};
typedef struct vertex_t
{
                struct V3 pos;
                struct V3 d;
                int n_tracks;
                int flags[2];
                struct track_t *tracks;
};
typedef struct track_t
{
                struct V3 start;
                struct V3 p;
                struct V3 d;
                struct V3 dp;
                float charge;
```

```
                float chisq;
                int n_hits;
                int flags[2];
                struct point_t *point_p;
                struct vertex_t *vertex_p;
                struct track_t *nextvt_p;
};
dsdb>  quit
examples>
```

This deals succesfully with the warnings from previous examples. Also when we ask for the definition of **track_t** we get the definitions of the objects it needs for the definitions to be complete. This listing may be used to make C header files.

4.3.5 Access to data

Following fortran subroutine accesses DSPACK data. It expects a structure of constants **POINT_C_T** to exist, copies it's contents into it's own structure and uses it to create 10 entries of structure **point_t**.

Example of access to data - point_fill.f file

```
      subroutine fill_points(ierr)
*
*    ******************************************************************
*    *                                                                *
*$$ *   subroutine fill_points  put some random data into 10 points    *
*    *                                                                *
*    ******************************************************************
*
      structure /V3/
         real  x
         real  y
         real  z
      end structure
      structure /point_t/
         record/V3/  pos
         record/V3/  d
         integer  flags(2)
         integer  next
      end structure
      record/point_t/ point(1)
      pointer ( point_p,point )

      structure/POINT_C_T/
         real A
         real B
         real C
         real D
      end structure
      record/POINT_C_T/ PC(2)

      data ivdsn_point/0/
      data ivdsn_const/0/
```

```
* Get constants into PC

      call dsget('POINT_C_T',ivdsn_const,2,nent,PC,ierr)
      if( nent.eq.0 ) return

* Tell DSPACK that right now we want pointers (remember what it was before)

      mode_old = ids_pnt_conv(1)

* Make 10 points

      point_p = idsput_ds('point_t',ivdsn_point,10)
      if( point_p.eq.0 ) then
         write(*,*)'Cannot make points'
         go to 995
      endif

* Build two link-lists of points:
* (1 3 5 7 9) and (2 4 6 8 10)

      z0 = 0.0
      do i=1,9,2
         point(i).pos.z = z0
         point(i).pos.y = PC(1).A*point(i).pos.z + PC(1).B
         point(i).pos.x = PC(1).C*point(i).pos.z + PC(1).D

         point(i).d.z = 0.01
         point(i).d.y = 0.01*point(i).pos.y
         point(i).d.x = 0.01*point(i).pos.x

         point(i).flags(1) = 1
         point(i).flags(2) = 2

         point(i).next = loc(point(i+2))
         z0 = z0 + 1.0
      enddo
      point(9).next = 0

      z0 = 0.0
      do i=2,10,2
         point(i).pos.z = z0
         point(i).pos.y = PC(2).A*point(i).pos.z + PC(2).B
         point(i).pos.x = PC(2).C*point(i).pos.z + PC(2).D

         point(i).d.z = 0.01
         point(i).d.y = 0.01*point(i).pos.y
         point(i).d.x = 0.01*point(i).pos.x

         point(i).flags(1) = 1
         point(i).flags(2) = 2

         point(i).next = loc(point(i+2))
         z0 = z0 + 1.0
      enddo
      point(10).next = 0
```

```
* Tell DSPACK that we have finished with point_t

      call ds_update('point_t',ivdsn_point,ierr)

* Reset pointer/index flag to what it was before

 995  continue
      idum = ids_pnt_conv(mode_old)

      end
```

Structure of constants **POINT_C_T** is defined and initialized in another object description file which should be used now instead of **dspack_tut_2.d**.

Object initialization - dspack_tut_4.d

```
*
*   *****************************************************************
*   *                                                               *
*$$ *   dspack_tut_4.d  DSPACK tutorial - example 2                 *
*   *                                                               *
*   *****************************************************************
*

DEFINE point_t S 1                 ! define structure point_t, section 1
                V3 pos             ! position
                V3 d               ! position errors
               .INT4  flags(2)
                point_t *next      ! pointer to the next hit on track
END DEFINE -CLASS point -TYPE 0

DEFINE vertex_t S 1                ! define structure point_t, section 1
                V3 pos             ! position
                V3 d               ! position errors
               .INT4 n_tracks      ! number of tracks
               .INT4  flags(2)
                track_t *tracks    ! pointer to tracks
END DEFINE

DEFINE POINT_C_T R 2               ! define point constants, section 2
               "Y slope"     A
               "Y intercept" B
               "X slope"     C
               "X intercept" D
END DEFINE

INIT POINT_C_T(1)    CREATE   2       ! Initialize two entries
      A = 0.01
      B = 1.0
      C = 0.02
      D = 0.5
END INIT

INIT POINT_C_T(2)
      A = 0.03
```

```
      B = 3.0
      C = {POINT_C_T(1).C}
      D = 2.5
END INIT
```

This file defines **POINT_C_T** as well as **point_t** and **vertex_t** and then proceeds to initialize **POINT_C_T**. Note how **POINT_C_T(2).C** is initialized in terms of another object. The definition of **point_t** has changed - it contains keywords setting a class of an object. Before this file is used we have to establish a new class of objects called **point**.

4.3.6 Using object classes

Subroutine fill_points (page 14) could be simply called from our program. However, a more general way of using it is to install it as a part of a **class driver** - this way it will be called automatically - when needed.

The entry routine for the class driver is shown below. It is called with a message and some information about the object.

point class driver routine - point_class.f

```
      subroutine point_class(msg,ivdsn,instr,idone)
*
*    ******************************************************************
*    *                                                                *
*$$ *   subroutine point_class  Driver of 'point' class               *
*    *                                                                *
*    ******************************************************************
*
      structure/instr_t/
        character*80 name
        integer class
        integer type
        integer lun
        integer pnt
      end structure
      record/instr_t/ instr
      character*(*) msg
      integer ivdsn,idone
      character*5 cid
      pointer (cid_p,cid)
      save

      if( msg.eq.'FILL' .or. msg.eq.'REFILL' ) then
        if( instr.name(:7).ne.'point_t' ) then
          idone = 0
        else
          call fill_points(ierr)
          if( ierr.eq.0 ) then
            idone = 1
          else
            idone = 0
          endif
        endif
```

```
      else if( msg.eq.'INIT' ) then
        idone = 1
      else if( msg.eq.'LIST' ) then
        write(instr.lun,'(''Point Class (CLASS=2000,IDENT=point)'')')
        idone = 1
      else if( msg.eq.'IDENT' ) then
        cid_p = instr.pnt
        cid = 'point'
        idone = 1
      else if( msg.eq.'NEW' ) then
        idone = 1
      else if( msg.eq.'PUT' ) then
        idone = 1
      else
        idone = 0
      endif

      end
```

This subroutine will be called when a message is sent to an object of class **point**. It calls **fill_points** independently whether the object has data or not when the message is sent (REFILL and FILL message). Just in case there is another object defined in the same class - there is a simple check whether the object in question is the one **fill_points** knows about. Other messages handled by the driver are the 'automatic' messages which DSPACK sends. A more general driver could react to other messages which the user can send to an object.

The driver has to be *installed*. This means that we have to pass the address of the driver's entry point to DSPACK. If our data description file is to be read on initialization, then the driver should be installed **before** DSPACK is initialized.

Example 2 - installing a class - file example_2.f

```
      program example_2
*
*   ******************************************************************
*   *                                                                *
*$$ *   program example_2  DSPACK tutorial example_2                 *
*   *                                                                *
*   *   Ryszard Zybert    Jul  1 12:58:10 1994                       *
*   ******************************************************************
*
      parameter ( IPCLASS=2000 )
      external point_class

* Install point class

      ipc = IPCLASS
      call ds_install(ipc,point_class)

* Initialise DSPACK, local mode, 2MB memory

      call dsinit(0,'example_2',6,500000,50000,ierr)

* Call ds_examine and do the rest interactively
```

```
      call ds_examine(' ')

      end
```

This program will use it's own settings from **dspackrc_2** file (page 9) because the name passed to dsinit changed to **example_2**.

The program is made using the same Makefile:

make example_2

The file **dspack_tut_4.d** should be used instead of **dspack_tut_2.d**. This can be done by using a different 'pilot' file.

dspack_tut_5.d file

```
*
*   ********************************************************************
*   *                                                                  *
*$$ *   dspack_tut_5.d  DSPACK tutorial - example 2                    *
*   *                                                                  *
*   ********************************************************************
*
INCLUDE dspack_tut_1.d
INCLUDE dspack_tut_4.d
```

Now everything is ready to run it. First we can look at the constants which were initialized in **dspack_tut_4.d**

Example 2 session

```
examples>example_2 tut_file=dspack_tut_5.d
 dspackrc_2 file in $DSPACK_HOME/doc/tutorial
 ********************
 * This is Example 2 *
 ********************
DSPACK 1.130,  5 Jul 1994
dsdb>  ls -l
Name             Type        Id     Sect    Size    Items   Nent    Bytes
V3                R          18       0       3       3       0        0
track_t           S          19       1      20      11       0        0
point_t           S          20       1       9       4       0        0
vertex_t          S          21       1      10       5       0        0
POINT_C_T         R          22       2       4       4       2       32

dsdb>  *POINT_C_T
 Data set: POINT_C_T
 Entry:     1
 Type:                  Real   (at:    7583408)
   1  A                 Y slope             : 1.00000E-02
   2  B                 Y intercept         : 1.00000
   3  C                 X slope             : 2.00000E-02
   4  D                 X intercept         : 0.500000
 Entry:     2
 Type:                  Real   (at:    7583424)
```

```
   1  A               Y slope          : 3.00000E-02
   2  B               Y intercept      : 3.00000
   3  C               X slope          : 2.00000E-02
   4  D               X intercept      : 2.50000
dsdb>
```

The **ls -l** command produces a long catalogue of objects. The first five columns deal with the definition of the object, the last two, with the data. **POINT_C_T** has two entries, taking 32 bytes of memory, other objects are empty. (V3 will always be empty - templates cannot keep data).

Although everything needed to fill **point_t** is in place, the object is still empty. Our program will be called as soon as a request for the data of **point_t** is made.

To start with - we can check if the driver was installed correctly.

Listing drivers

```
dsdb> drivers
Point Class (CLASS=2000,IDENT=point)
Calculator Class (CLASS=10,IDENT=calc)
Environment Class (CLASS=20,IDENT=env)
Hepdb Class (CLASS=40,IDENT=hepdb)
Exec Class (CLASS=50,IDENT=exec)
dsdb>
```

Now we can trigger the driver's action by just looking at **point_t** data.

Class driver in action

```
dsdb> ls -l
Name             Type        Id     Sect    Size    Items   Nent    Bytes
V3                 R         18       0       3       3       0        0
track_t            S         19       1      20      11       0        0
point_t            S         20       1       9       4       0        0
vertex_t           S         21       1      10       5       0        0
POINT_C_T          R         22       2       4       4       2       32

dsdb> *point_t(3:5)
 Data set: point_t(3:5)
 Entry:     3
 Type:                  Structure   (at:    5704996)
   1  pos               V3
        Type:                  Real   (at:    5704996)
          1  x                 X component        : 0.520000
          2  y                 Y component        : 1.01000
          3  z                 Z component        : 1.00000
   2  d                 V3
        Type:                  Real   (at:    5705008)
          1  x                 X component        : 5.20000E-03
          2  y                 Y component        : 1.01000E-02
          3  z                 Z component        : 1.00000E-02
   3  flags(1)          Integer        : 1
   3  flags(2)          Integer        : 2
   4  *next             Pointer        : 5705068
```

```
 Entry:     4
 Type:                  Structure   (at:    5705032)
   1  pos               V3
       Type:                   Real   (at:    5705032)
         1  x                  X component        : 2.52000
         2  y                  Y component        : 3.03000
         3  z                  Z component        : 1.00000
   2  d                 V3
       Type:                   Real   (at:    5705044)
         1  x                  X component        : 2.52000E-02
         2  y                  Y component        : 3.03000E-02
         3  z                  Z component        : 1.00000E-02
   3  flags(1)          Integer        : 1
   3  flags(2)          Integer        : 2
   4  *next             Pointer        : 5705104
 Entry:     5
 Type:                  Structure   (at:    5705068)
   1  pos               V3
       Type:                   Real   (at:    5705068)
         1  x                  X component        : 0.540000
         2  y                  Y component        : 1.02000
         3  z                  Z component        : 2.00000
   2  d                 V3
       Type:                   Real   (at:    5705080)
         1  x                  X component        : 5.40000E-03
         2  y                  Y component        : 1.02000E-02
         3  z                  Z component        : 1.00000E-02
   3  flags(1)          Integer        : 1
   3  flags(2)          Integer        : 2
   4  *next             Pointer        : 5705140
dsdb> ls -l
Name             Type        Id     Sect     Size     Items    Nent     Bytes
V3                R          18       0        3        3        0         0
track_t           S          19       1       20       11        0         0
point_t           S          20       1        9        4       10       360
vertex_t          S          21       1       10        5        0         0
POINT_C_T         R          22       2        4        4        2        32

dsdb>
```

The first **ls -l** shows no entries for **point_t**. However, the request for display of data causes a **FILL** message to be sent to the object - our driver is called and fills the object with data. It can be seen that our link-list seems to be contructed properly - point_t(3) has a pointer which agrees with the address of point_t(5). This can be seen easily by asking DSPACK to return indexes rather than pointers:

Displaying index values

```
dsdb> indexes
Showing indexes
dsdb> *point_t(3:5)
 Data set: point_t(3:5)
 Entry:     3
 Type:                  Structure   (at:    5704996)
   1  pos               V3
```

```
       Type:                 Real   (at:    5704996)
         1  x                X component        : 0.520000
         2  y                Y component        : 1.01000
         3  z                Z component        : 1.00000
   2  d               V3
       Type:                 Real   (at:    5705008)
         1  x                X component        : 5.20000E-03
         2  y                Y component        : 1.01000E-02
         3  z                Z component        : 1.00000E-02
   3  flags(1)        Integer        : 1
   3  flags(2)        Integer        : 2
   4  *next           Index          : 5
 Entry:     4
 Type:                Structure   (at:    5705032)
   1  pos             V3
       Type:                 Real   (at:    5705032)
         1  x                X component        : 2.52000
         2  y                Y component        : 3.03000
         3  z                Z component        : 1.00000
   2  d               V3
       Type:                 Real   (at:    5705044)
         1  x                X component        : 2.52000E-02
         2  y                Y component        : 3.03000E-02
         3  z                Z component        : 1.00000E-02
   3  flags(1)        Integer        : 1
   3  flags(2)        Integer        : 2
   4  *next           Index          : 6
 Entry:     5
 Type:                Structure   (at:    5705068)
   1  pos             V3
       Type:                 Real   (at:    5705068)
         1  x                X component        : 0.540000
         2  y                Y component        : 1.02000
         3  z                Z component        : 2.00000
   2  d               V3
       Type:                 Real   (at:    5705080)
         1  x                X component        : 5.40000E-03
         2  y                Y component        : 1.02000E-02
         3  z                Z component        : 1.00000E-02
   3  flags(1)        Integer        : 1
   3  flags(2)        Integer        : 2
   4  *next           Index          : 7
dsdb>
```

There is also a special syntax which can be used to display a link list. One has to specify the pointer to the list and the name of follow-up pointer, separated by a double colon:
The link-list must be terminated by a NULL pointer.

Displaying link-lists

```
dsdb> pointers
Showing pointers
dsdb> *point_t(2).*next::*next
 Data set: point_t(2).*next::*next,   Entry:      2
 Link-list entry:     1
```

```
Entry:       4 out of      10
Type:                 Structure   (at:    5705032)
  1  pos              V3
      Type:                 Real   (at:    5705032)
        1  x                X component        : 2.52000
        2  y                Y component        : 3.03000
        3  z                Z component        : 1.00000
  2  d                V3
      Type:                 Real   (at:    5705044)
        1  x                X component        : 2.52000E-02
        2  y                Y component        : 3.03000E-02
        3  z                Z component        : 1.00000E-02
  3  flags(1)         Integer       : 1
  3  flags(2)         Integer       : 2
  4  *next            Pointer       : 5705104
Link-list entry:     2
Entry:       6 out of      10
Type:                 Structure   (at:    5705104)
  1  pos              V3
      Type:                 Real   (at:    5705104)
        1  x                X component        : 2.54000
        2  y                Y component        : 3.06000
        3  z                Z component        : 2.00000
  2  d                V3
      Type:                 Real   (at:    5705116)
        1  x                X component        : 2.54000E-02
        2  y                Y component        : 3.06000E-02
        3  z                Z component        : 1.00000E-02
  3  flags(1)         Integer       : 1
  3  flags(2)         Integer       : 2
  4  *next            Pointer       : 5705176
Link-list entry:     3
Entry:       8 out of      10
Type:                 Structure   (at:    5705176)
  1  pos              V3
      Type:                 Real   (at:    5705176)
        1  x                X component        : 2.56000
        2  y                Y component        : 3.09000
        3  z                Z component        : 3.00000
  2  d                V3
      Type:                 Real   (at:    5705188)
        1  x                X component        : 2.56000E-02
        2  y                Y component        : 3.09000E-02
        3  z                Z component        : 1.00000E-02
  3  flags(1)         Integer       : 1
  3  flags(2)         Integer       : 2
  4  *next            Pointer       : 5705248
Link-list entry:     4
Entry:      10 out of      10
Type:                 Structure   (at:    5705248)
  1  pos              V3
      Type:                 Real   (at:    5705248)
        1  x                X component        : 2.58000
        2  y                Y component        : 3.12000
        3  z                Z component        : 4.00000
  2  d                V3
      Type:                 Real   (at:    5705260)
```

```
        1  x                X component         : 2.58000E-02
        2  y                Y component         : 3.12000E-02
        3  z                Z component         : 1.00000E-02
   3  flags(1)          Integer        : 1
   3  flags(2)          Integer        : 2
   4  *next             Pointer        : 0
dsdb>
```

Note that the list above starts from **point_t(4)** which is the second element of the real list. In order to display the complete list we would need a pointer to the first element of the list - we don't have it yet, so we use the pointer stored in **point_t(2)** which points to **point_t(4)**.

4.3.7 Using calculator

Calculator is installed as a built-in class in DSPACK. The following examples demonstrate how the calculator may be used interactively. If the object we want to calculate is undefined - the standard definition template is used. The expression is stored as an object resource and the object is assigned to **calculator** class. If the new object is defined, it's type is decided by the type of the result, otherwise the result is converted to the type needed.

Calculate mean of all entries

```
dsdb> calc mean_y = SUM(point_t.pos.y)/NENT(point_t)
dsdb> mean_y
 mean_y = SUM(point_t.pos.y)/NENT(point_t)
 Definition of data set: mean_y             (mean) ID:    23
 Type:                   Real
 Section:                    1
 Machine words:              1
 Items:                      1
   1  CALC

typedef struct mean_y
{
                float CALC;
};
dsdb> ls -l
Name            Type        Id     Sect    Size    Items   Nent    Bytes
V3                R         18       0       3       3       0        0
track_t           S         19       1      20      11       0        0
point_t           S         20       1       9       4      10      360
vertex_t          S         21       1      10       5       0        0
POINT_C_T         R         22       2       4       4       2       32
mean_y            R         23       1       1       1       1        4

dsdb> *mean_y
 Data set: mean_y
 Entry:     1
 Type:                  Real   (at:    5705336)
   1  CALC                              : 2.04000
```

In this case the 10 entries of **point_t** result in a single number. Usually the result is a vector, like in the following example.

Example - Calculating distance from Z axis

```
dsdb> calc r_point = SQRT(point_t.pos.x^2+point_t.pos.y^2)
dsdb> *r_point.CALC
    1  r_point.CALC  =        1.11803
    2  r_point.CALC  =        3.90512
    3  r_point.CALC  =        1.13600
    4  r_point.CALC  =        3.94098
    5  r_point.CALC  =        1.15412
    6  r_point.CALC  =        3.97683
    7  r_point.CALC  =        1.17239
    8  r_point.CALC  =        4.01269
    9  r_point.CALC  =        1.19080
   10  r_point.CALC  =        4.04855
dsdb>
```

If we use 'logical' expression, we get 'logical' result:

Example - Logical expressions

```
dsdb> calc distant = r_point.CALC>2
dsdb> distant
 distant = r_point.CALC>2
 Definition of data set: distant            (dist) ID:    25
 Type:                   Logical
 Section:                    1
 Machine words:              1
 Items:                      1
   1  CALC

typedef struct distant
{
                int CALC;
};
dsdb> *distant.CALC
    1  distant.CALC =     F
    2  distant.CALC =     T
    3  distant.CALC =     F
    4  distant.CALC =     T
    5  distant.CALC =     F
    6  distant.CALC =     T
    7  distant.CALC =     F
    8  distant.CALC =     T
    9  distant.CALC =     F
   10  distant.CALC =     T
dsdb>
```

Calculator objects are filled with data following the stored expressions. In all other respects they behave like all other objects.

In order to perform the calculations the calculator driver makes requests for all source objects needed. This means that - the **FILL** message will be sent to **point** driver (or **REFILL** if **point_t** has data. It is demonstrated in the following example, where **point_t** is filled with data following a message from **calculator**. Note **Nent** column for **point_t**.

Example - indirect filling of objects

```
dsdb> ls -l
Name            Type        Id     Sect     Size     Items    Nent     Bytes
V3               R          18       0        3        3        0         0
track_t          S          19       1       20       11        0         0
point_t          S          20       1        9        4        0         0
vertex_t         S          21       1       10        5        0         0
POINT_C_T        R          22       2        4        4        2        32

dsdb> calc mean_y = SUM ( point_t.pos.y ) /NENT ( point_t )
dsdb> ls -l
Name            Type        Id     Sect     Size     Items    Nent     Bytes
V3               R          18       0        3        3        0         0
track_t          S          19       1       20       11        0         0
point_t          S          20       1        9        4        0         0
vertex_t         S          21       1       10        5        0         0
POINT_C_T        R          22       2        4        4        2        32
mean_y           R          23       1        1        1        0         0

dsdb> *mean_y
 Data set: mean_y
 Entry:     1
 Type:                 Real   (at:    5705320)
   1  CALC                               : 2.04000
dsdb> ls -l
Name            Type        Id     Sect     Size     Items    Nent     Bytes
V3               R          18       0        3        3        0         0
track_t          S          19       1       20       11        0         0
point_t          S          20       1        9        4       10       360
vertex_t         S          21       1       10        5        0         0
POINT_C_T        R          22       2        4        4        2        32
mean_y           R          23       1        1        1        1         4

dsdb>
```

In the process of calculating **mean_y point_t** has been filled with data.

4.3.8 Pointers and Indexes

DSPACK handles pointers by converting them into indexes - **entry numbers** when necessary - for instance for I/O. This feature may be used by the programmer for cases when it is more convenient to use indexes rather than pointers.

One example of this was presented in the example session on page 21 where the user can select whether pointers or indexes are displayed.

This feature can also be used from a program. We can modify the program **fill_points** (page 14) so that we can store indexes of the link-list instead of pointers.

Modifying subroutine fill_points to use indexes

```
WE HAVE TO REPLACE FOLLOWING LINES:

      mode_old = ids_pnt_conv(1)    ! 1 means ´use pointers´
```

```
should be replaced by:

      mode_old = ids_pnt_conv(2)    ! 2 means ´use indexes´

And inside both loops the lines:

         point(i).next = loc(point(i+2))

should be replaced by:

         point(i).next = i+2
```

The resulting program will produce data functionally identical to the original one.

Part III

DSPACK – User Guide

Part IV

DSPACK – Advanced Features

DSPACK - Advanced Features

User's Guide
(Version 1.128, preliminary)

Ryszard Zybert
School of Physics and Space Research
University of Birmingham

22^{nd} June 1994

Table of Contents

Chapter 5: Introduction

Basic function of DSPACK is to help the user to store and manipulate data. This can all be achieved by the set of simple functions described in the User's Guide.

Features described here are extensions which may be used in order to make the system easier to use and more flexible. Some of the features allow mapping of data from other systems into DSPACK objects, other extend power of DSPACK.

In some cases using the features described here may mean more work for the programmer - in order to make life easier for the user. In the High Energy Physics community, where more and more of us become programmers as well as users - this approach means investing into making **our** life easier at a later date.

Chapter 6: Object Classes

DSPACK allows the user to use concepts of Object Oriented programming with dynamically defined objects.

In Object Oriented programming an object belongs to a **Class** which defines the behaviour or the object. The operations on the object are performed by sending a **message** to the object - each message is handled by a **method** which is characteristic for the class.

6.1 Why Object Oriented programming?

The concept of objects containing data is used in High Energy Physics for many years. Programs use **banks** to keep results of some calculations, the data are stored in **histograms**, **ntuples**, etc. Object oriented programming moves this technique one step further. Not only we keep data in **named** objects. We also make sure that the information about the operations on objects and even the instructions on how to fill the object with data is kept in the object's definition.

Traditionally - whether we used objects to keep data or not - the processing bewteen the raw data and the final result consisted of a series of sequential steps - each of them - like a filter - converting data from one form into another, often combining results from several earlier steps. As the experiments get larger, the collaborations get larger, the number of people who independently contribute to the final analysis gets larger - this process is becoming very complicated. Each step of the processing chain is dependent on other steps and very few of us have expert knowledge of details of more than a few of the steps. The result is that - as users of the software - we are often facing the situation where it is difficult to get the objects we want because they depend on other objects which require extra programs - which in turn depend on something else... and so on.

Object oriented programming goes some way to get this problem under control. By allowing objects to contain more information about themselves and providing a system capable of using this information - we should achieve the situation where the user will be able to request an object he/she wants - without worrying what other objects may be necessary in order to satisfy the request - and what other programs need to run.

Additional advantages of Object Oriented Programming are not less important in practice. The number of systems used to keep data is very large. We are faced all the time with a need to write interfaces between the systems. Object Oriented Programming provides a natural way of making such interfaces and making them transparent to the user. The object of class 'A' will then behave like any other object, although - behind the scene the interface 'A' has been called in order to convert the object from/to it's original form.

More obvious feature of Object Oriented Programs is the fact that the objects may have very different contents and structure and the operations on these very different objects may be handled in a simple and elegant way (for instance - **displaying** a histogram is a very different action from **displaying** a track - but sending a **DISPLAY** message to these two objects should produce the effect expected by the user)

6.2 Basic Terms

`Polymorphism`	ability of the system to handle the same message in a different way for different object
`Message`	a 'command' from the program to the object
`Method`	The code which is executed in response to a message
`Inheritance`	Ability of the object (or a class) to 'inherit' methods of another class (it's **superclass**) - so that the class does not need to implement methods for all possible messages.

DSPACK implements these concepts in the following manner:

A class (or a set of classes) for each object may be set after the object has been defined. All methods required for the class are kept in a **Class Driver** - a program with a single entry point which is called when the program sends a message to the object of that class. Usually the program must be linked with the user's code and DSPACK should be informed with a **ds_install** call (but there are exceptions where this is not required).

Together with the class of an object the user may specify object **type** - this is a small integer number which is never used by DSPACK, but is passed together with the message - allowing the driver to handle 'sub-classes' of it's own class.

The driver may respond to the message if it contains a method for it or it can ignore it. If the message is ignored by the driver (or if it 'pretends' to ignore it - see below) then the next class up of the 'genealogical tree' of the object is called and so on - until the message is handled.

In some cases it may be convenient for the driver to perform some action in response to the message but still allow inherited methods to work - the driver may 'pretend' not to handle the message. (An example: DSPACK contains a standard method to show the contents of an object - in response to a **SHOW** message. If the object belongs to a class, then the class driver is called first and standard SHOW is performed only if the driver does not respond - this allows the driver to show extra information about the object and then force the standard method to be used for the contents display)

6.3 Automatic messages

Messages may be sent to the object by user's program, they may also be passed on by interactive tools. However, some messages are sent to objects by DSPACK. The current list of 'automatic' messages is as follows:

`SHOW`	interactive program requested a display of object contents.
`SHOWDEF`	interactive program requested a display of object definition
`FILL`	contents of the object has been requested and the object has no data
`REFILL`	contents of the object has been requested and the object has data
`PUT`	The object has been filled or updated
`NEW`	The object class has just been assigned

6.4 Class Drivers

All methods defined by a class should be accessible through one entry point, called **Class Driver**. The calling sequence of the class driver is as follows:

```
call driver  (msg,ivdsn,instr,idone)
```

`msg`	Character variable, NULL terminated, containing the message
`ivdsn`	Object handle
`instr`	A structure of extra information - see below
`idone`	Integer flag returned by the driver - 0 means 'not-done', anything else means done

instr is a structure which allows the caller and the driver to pass extra information.

Driver Call Structure

```
structure/instr_t/
  character*16 name    ! Object name
  integer class        ! Object class
  integer type         ! Object type (sub-class)
  integer lun          ! Fortran unit number for possible output
  integer pnt          ! Pointer for returned data
end structure
```

This form of the input structure allows the caller to request extra information to be returned via pnt.

6.4.0.1 Messages to the driver

The communication between the program and the driver allows the driver to be called with special messages - and no object. The current list of **obligatory** messages the driver must respond to is as follows:

INIT	Driver initialisation. If the driver is installed before DSPACK initialisation - then another INIT call will be sent after DSPACK initialisation. The driver may check whether DSPACK has been initialised by calling **integer function ids_done(ierr)** - it returns 1 if DSPACK has been initialised and 0 otherwise.
LIST	Driver should write it's identification message on unit lun - used to list installed drivers.
IDENT	Identification request. pnt is a pointer to a character variable and the driver should copy it's name into that variable. This is the only mechanism available for the system to associate the class name with class number.

6.4.0.2 Examples

EXEC driver

```
      subroutine ds_exec_dr(msg,ivdsn,instr,idone)
*
*   ******************************************************************
*   *                                                                *
*$$ *   subroutine ds_exec_dr  Driver of exec class                  *
*   *                                                                *
*   *   Ryszard Zybert    May 26 16:02:47 1994                       *
*   ******************************************************************
*
      structure/instr_t/
        character*16 name
        integer class
        integer type
        integer lun
        integer pnt
      end structure
      record/instr_t/ instr
      character*(*) msg
      integer ivdsn,idone
      character*4 cid
      pointer (cid_p,cid)
      save

      if( msg.eq.'INIT' ) then
        if( ids_done(ierr).gt.0 ) then
          call ds_execdr_init
          idone = 1
        else
          idone = 0
        endif
      else if( msg.eq.'LIST' ) then
        write(instr.lun,'(''Exec Class (CLASS=50,IDENT=exec)'')')
        idone = 1
      else if( msg.eq.'IDENT' ) then
        cid_p = instr.pnt
        cid = 'exec'
        idone = 1
      else
        call ds_execdr_fill(instr.type,msg,idone)
      endif

      end
```

CALC driver

```
#include "dspack_cl.h"
typedef struct instr_t {
    char name[16];
    int class;
    int type;
    int lun;
    char *ident;
```

```
};

/*
*   ****************************************************************
*   *                                                              *
*$$ *   void ds_calc_dr_  calculator class driver                  *
*   *   Input:                                                     *
*   *         comm - character - command                           *
*   *         ivdsn - integer - object handle                      *
*   *         instr - struct - input structure                     *
*   *    Output:                                                   *
*   *         idone - integer - ´done´ flag                        *
*   *                        0 if not done                         *
*   *                        1 if done                             *
*   *                                                              *
*   *   Ryszard Zybert    Apr 14 16:21:13 1994                     *
*   ****************************************************************
*/
#define REFILL_TYPE 2

void ds_calc_dr_(comm,ivdsn,instr,idone,len)
char *comm;
int *ivdsn;
struct instr_t *instr;
int *idone;
int len;
{
      int ierr;
      char coml[80];
      static int last_iv;

      *idone = 0;
      if ( instr->class==CALCULATOR_CLASS )
      {
        memcpy(coml,comm,len);
        coml[len] = ´\0´;
        if ( str_equal(coml,"FILL") ||
           ( str_equal(coml,"REFILL") && instr->type==REFILL_TYPE ) )
        {
          if( *ivdsn!=last_iv )             /* Don´t allow recursive calls here */
          {
            last_iv = *ivdsn;
            ds_calc_dr_fill_(ivdsn);
            last_iv = 0;
            *idone = 1;
          }
        }
        else if ( str_equal(coml,"INIT") )
        {
          last_iv = 0;
          *idone = 1;
        }
        else if ( str_equal(coml,"LIST") )
        {
          if ( ids_done_(&ierr) )
            printf( "Calculator Class (CLASS=10,IDENT=calc)\n" );
          *idone = 1;
```

```
        }
        else if ( str_equal(coml,"IDENT") )
        {
          strcpy(instr->ident,"calc",4);
          *idone = 1;
        }
      }
}
```

6.5 Installing a Class

There are three distinct modes of operation with class drivers:

`Local` The class driver is a part of the user's program. This is independent of whether the user uses DSPACK server or not. The driver must be linked with the user's program and DSPACK should be informed with a **ds_install** call.

`Stand-Alone` In the client-server operation the class methods may be a separate process. In this mode this process should be started once and then called whenever there is a message to it's object. This is achieved by installing the **interface** driver in the user's program. The process containing methods is started by the interface driver when INIT message is received. Afterwards the interface driver may communicate with the methods process in any way it wants to - DSPACK provides an easy way of communication with **ids_wait** and **ds_update** calls which allow a client to wait for an object to become available.

`EXEC` In the client-server operation the program containing methods may be started every time a message is sent to an object of it's class. The program executes the method requested and exits. This mode requires no installation and no driver. The program must assume that all needed objects are in the server's memory, it connects to the server and fulfils the request. See EXEC driver description for details.

call ds_install `(iclass,driver)`

`iclass` Integer - class number. At present class numbers are assigned manually. Each class should have a unique number between 1000 and 32000. Classes from 1 to 999 are reserved for DSPACK internal drivers.

`driver` Driver entry point - must be declared external.

In **Local** or **Stand-Alone** mode this call installs a class driver.

6.6 Pre-installed Classes

At present there are 4 built-in class drivers in DSPACK. This number will hopefully increase quickly. There are about 20 proposed classes which sooner or later will become a part of DSPACK library.

6.6.1 Calculator

calc)

Calculator objects contain an expression which tells the program how their values can be obtained from other DSPACK objects.

The user can program the calculator object by typing the expression interactively, placing it in the **Object Description File** or by calling

```
call dscprg  (expression,ivdsn,ierr)
```

`expression`	Calculator expression in the form: **name = formula**. Calculator can distinguish between **Algebraic Notation** and **Inverse Polish Notation** and both are legal.
`ivdsn`	Object handle - if 0 the correct handle will be returned
`ierr`	Error flag - should be 0

In all cases the class of the object is set to 'calc' and the **type** is set to 0.

Calculator object types:

`type=0`	For these objects the calculator driver only reacts to **FILL** message. This means that the object will be calculated only if it contains no data. Subsequent accesses will just return the values.
`type=2`	Calculator reacts to **FILL** and **REFILL** messages. These objects will be recalculated every time they are accessed.

Examples:

The first example shows how all items of an object can be calculated from another object. The calculation has to be performed only once.

Simple calculator expressions

```
CALC VTP2_DIMS.FIRST_ROW = VTP2_SECTORS.ZC1-VTP2_SECTORS.DZ/2+VTP2_SECTORS.ZBORDER
CALC VTP2_DIMS.ROW_STEP  = (VTP2_SECTORS.DZ-2*VTP2_SECTORS.ZBORDER)/VTP2_SECTORS.NROWS
CALC VTP2_DIMS.YMAX      = VTP2_SECTORS.YC+VTP2_SECTORS.DY/2
CALC VTP2_DIMS.XMAX      = VTP2_SECTORS.XC2+VTP2_SECTORS.DX/2
CALC VTP2_DIMS.XMIN      = VTP2_SECTORS.XC2-VTP2_SECTORS.DX/2
```

The following example shows a series of calculations needed to find a mean value of **pr_v1_guess.z** and it's error. The calculation should be repeated once per event in order for the sums to be calculated. The object should be stored in section 2 (constants) and accessed **exactly** once per event.

Calculator objects of type=2 - an example

```
DEFINE v1 S 2
        .REAL4  sum
        .REAL4  ss
        .INT4   n
        .REAL4  z
        .REAL4  dz
END DEFINE

CALC v1.sum = v1.sum + pr_v1_guess.z
CALC v1.ss  = v1.ss  + pr_v1_guess.dz^2
CALC v1.n   = v1.n   + 1
CALC v1.z   = v1.sum/v1.n
CALC v1.dz  = SQRT(v1.ss)/v1.n

RECLASS v1 -CLASS calc -TYPE 2  ! need to RECLASS - the default is TYPE=0

INIT v1
   sum = 0.0
   ss  = 0.0
   n   = 0
END INIT
```

All calculations are performed on ALL entries of the source objects and the result will have the same number of entries as the source. In each expression all objects must either have the same number of entries or **ONE** entry - the element with one entry is treated as a constant and it's value is extended - repeated for all entries of other objects.

Calculator functions.

The full list of calculator functions is as follows:

Calculator functions

```
+
-
*
/
SQRT
LOG (ln)
SQRE (x**2)
EXP
XtoY (X**Y)(X^Y)
ABS
NEG (-x)
SIN
COS
TAN
ASIN
ACOS
ATAN
1/X
INT
MAX (2 arguments)
MIN (2 arguments)
RAN (argument * random number (0:1))
```

```
PROB(X,N) (2 arguments)   (CERNLIB PROB - G100 )
SUM - the sum of all entries
NENT - number of entries
AND
OR
XOR - (exclusive or)
NOT
>  (gt)
<  (lt)
=  (eq)
!= (ne)
>= (ge)
<= (le)
IND(N,X) - Nth entry of X
TENT - number of true entries
```

If the calculator expression is programmed for an existing object, the result is converted to the type of the target object.

The logical functions produce logical results - if converted to integers or reals they are 0 for FALSE and -1 for TRUE.

Functions **SUM**, **NENT** and **TENT** result in a single entry when applied to multi-entry operands. If these functions are used as a part of larger expression - they behave like constants - they are repeated for each entry of the result. For **IND(N,X)** the number of entries in **N** determines the number of entries of the result.

6.6.2 Environment Class

Environment class provides easy access to UNIX environment variables.

Example of Environment Object

```
DEFINE Etest S 2
      .CHAR16 USER
      .INT4   WINDOWID
END DEFINE -CLASS env -TYPE 0
```

Etest.USER will always contain the contents of $USER variable and Etest.WINDOWID will return $WINDOWID - if available, or 0 otherwise.

There are three types of **env** objects:

TYPE=0 The substitution is made every time the object is requested. If the environment variable is not set, corresponding item is set to 0.

TYPE=1 The substitution is made every time the object is requested. If the environment variable is not set then the user is prompted to supply the value on the first access. All subsequent accesses use the value supplied.

TYPE=2 The substitution is made every time the object is requested. If the environment variable is not set then the user is prompted to supply the value.

6.6.3 EXEC class

EXEC class is mainly used in Server/Client operation. All methods are stand-alone programs or shell commands. The driver allows the user to specify up to 20 sub-classes and in ech of them up to 5 different commands for handling different messages for each sub-class.

The driver creates an object *.exec*:

Definition of EXEC sub-classes

```
      structure /exec_method/
              character*80  command
              character*80  messages
      end structure
      structure /exec/
              record/exec_method/  methods(5)
      end structure
```

`command` System command to be executed

`messages` List of messages (separated by spaces ot commas) which triggers the command

20 entries of *.exec* are created when the driver is initialized. The user can fill any of them with required action.

Example of EXEC sub-class definition

```
INIT .exec(1)                                 ! CLASS exec TYPE 1
     methods(1).command  = dsread >/dev/null
     methods(1).messages = FILL
END INIT

INIT .exec(2)                                 ! CLASS exec TYPE 2
     methods(1).command  = $PATREC_HOME/bin/patrec_ds >/dev/null
     methods(1).messages = FILL
END INIT
```

The definitions above ensure that an attempt to access any object of class exec and type 1 will trigger **dsread** if the object has no data. And the attampt to access an object of class exec and type 2 will run **patrec_ds**. The process is recursive - so, if patrec_ds requires objects which are of CLASS exec and type 1 - **dsread** will automatically be called (unless the object is already there).

6.6.4 (HEPDB class)

Manual in preparation

Chapter 7: Object Description Files

DSPACK objects may be defined by direct calls, by reading DSPACK data files or by reading Object Description Files. Object description file is an ASCII file which may be edited by the user. The 'free' format of the file helps in designing the files in such a way that they may be used for documentation.

7.1 Rules

Empty lines are ignored.

Lines starting from a '*' or a '#' are ignored.

In-line comments starting from ' !' are ignored.

'-' or '\' at the end of the line is a continuation character.

If a line does not start from a keyword and contains an equal sign - it is interpreted as an assignment line.

7.2 Syntax

SET `name value`

`name` alias name
`value` replacement text

each occurrence of name in the rest of the file will be substituted by value. (similar to '#define' in cpp)

INCLUDE `file`

`file` file name

include another description file here

DEFINE object object-type section
item-type name
item-type name
item-type name
...
END DEFINE [-CLASS class [-TYPE type]]

`object`	object name
`object-type`	object type (I,R,S,L,C - see DSDEF)
`section`	section - 1 for variables, 2 for constants
`item-type`	If object-type is S, then item-type must be a name of existing DSPACK object. Otherwise it may be any character string describing the item.
`name`	Item name. (max. 16 characters). If the item is a pointer, the name must start from a '*'. If the item is an array, the size is specified in brackets (name(n))

`class`	Object class name. The inheritance may be specified here in the following way: if the class is specified as class1.class2...classn then the object is assigned to class "classn" and the methods are inherited from all classes specified (in the reverse order - class(n-1) is the parent, class(n-2) is a grandparent etc.
`type`	sub-class - small integer passed to the all class drivers, ignored by DSPACK

or

DEFINE `object section LIKE object1 [-CLASS class [-TYPE type]]`

Defines DSPACK object. The second form uses object1 definition as a template. The template is used but not copied - this means that this form actually saves memory needed to store object definitions.

CREATE `object number-of-entries`

creates a number entries of an object. All existing entries are destroyed.

name `= value`

Assignment. name must be the full name of a specific DSPACK item, may contain index if necessary. value may have following forms:

`literal`	a number or character string
`{name}`	name of another DSPACK item
`$name`	name of environment variable

If an assignment is made to an object which has no data at that point - the object is created to contain the necessary number of (uninitialized) entries so that the assignment can be made. If the object exist but the number of entries is smaller than needed for the assignment, an error is reported and nothing is assigned. Therefore if a multi-entry object is to be initialised this way CREATE should be used (or the first assignment should be made of the highest entry needed).

INIT prefix [CREATE nent]
suffix = value
suffix = value
suffix = value
...
END INIT

Another form of assignment. Each item name is constructed by joining prefix and suffix into complete item name (prefix.suffix). All rules of a normal assignment apply.

object `value value value...`

Yet another form of assignment. The whole object (single entry) if filled by this statement. Values have to be literal in this case. If character variables contain blanks they may be quoted.

CALC `name = formula`

Definition of calculator object. Two forms are allowed:

- If name is an item of a previously defined object the formula is attached to this object and the class of the object is set to CALC, TYPE of the object is set to 0.
- If name is not found (and does not contain dots) then a new object is defined with one word called CALC. The type of the word is appropriate for the result of formula. The class of the object is CALC, type is 0.

formula is algebraic or inverse polish notation formula involving defined DSPACK objects. See page 9calc).

RECLASS `object -CLASS class -TYPE type`

Change class of an object - see DEFINE for details.

Object Description File - Example

```
######################### patrec parameter file ########################
#
#   *****************************************************************
#   *                                                               *
#$$ *   patrec.d  NA49 Vtpc2 pattern recognition constants.         *
#   *   This file contains definitions and values for PATREC constants *
#   *   The modifications should be commented and .PATREC_PAR_VER     *
#   *   should be modified to contain correct data and version number. *
#   *   DEFINE .PATREC_PAR_VER R 2                                  *
#   *                                                               *
#   *   Ryszard Zybert    Jun 23 17:34:26 1994                      *
#   *****************************************************************
#
DEFINE .PATREC_PAR_VER R 2
        Version       VERSION
        Date          DATE
END DEFINE

.PATREC_PAR_VER                     1.200              940517.0

DEFINE POINT R 0                       ! a template only
        'X coordinate' X
        'Y coordinate' Y
        'Z coordinate' Z
END DEFINE

DEFINE vt2_hit S 1                     ! internal PATREC hits storage
   POINT    pos
   .REAL4   ex
   .REAL4   ey
   .INT4    row
END DEFINE

DEFINE vt2_hitp I 1                    ! list of original hit pointers
        'NASE pointer' pointer
```

```
END DEFINE

DEFINE VTP2_SECTORS S 2                ! VTPC2 geometry
  .REAL4    DX
  .REAL4    DY
  .REAL4    DZ
  .REAL4    XC1
  .REAL4    XC2
  .REAL4    YC
  .REAL4    ZC1
  .REAL4    ZC2
  .REAL4    ZC3
  .REAL4    ZBORDER
  .INT4 NROWS
END DEFINE

VTP2_SECTORS.DX       = 72.0
VTP2_SECTORS.DY       = 70.0
VTP2_SECTORS.DZ       = 72.0
VTP2_SECTORS.XC1      = -47.1
VTP2_SECTORS.XC2      = 47.1
VTP2_SECTORS.YC       = 0.0
VTP2_SECTORS.ZC1      = -77.0
VTP2_SECTORS.ZC2      = 0.0
VTP2_SECTORS.ZC3      = 77.0
VTP2_SECTORS.ZBORDER  = 2.00
VTP2_SECTORS.NROWS    = 24


DEFINE VTP2_DIMS S 2                  ! geometry the way we want it
  .REAL4   FIRST_ROW                  ! this is calculated from
  .REAL4   ROW_STEP                   ! VTP2_SECTORS
  .REAL4   YMAX
  .REAL4   XMAX
  .REAL4   XMIN
END DEFINE

CALC VTP2_DIMS.FIRST_ROW = VTP2_SECTORS.ZC1-VTP2_SECTORS.DZ/2+VTP2_SECTORS.ZBORDER
CALC VTP2_DIMS.ROW_STEP  = (VTP2_SECTORS.DZ-2*VTP2_SECTORS.ZBORDER)/VTP2_SECTORS.NROWS
CALC VTP2_DIMS.YMAX      = VTP2_SECTORS.YC+VTP2_SECTORS.DY/2
CALC VTP2_DIMS.XMAX      = VTP2_SECTORS.XC2+VTP2_SECTORS.DX/2
CALC VTP2_DIMS.XMIN      = VTP2_SECTORS.XC2-VTP2_SECTORS.DX/2

DEFINE VTP2_L3_PAR  R 2              ! Tolerances: - for 3 hits
   'X tolerance' XTOL
   'Y tolerance' YTOL
END DEFINE
DEFINE VTP2_L4_PAR  2 LIKE VTP2_L3_PAR    ! for 4 hits
DEFINE VTP2_L5_PAR  2 LIKE VTP2_L3_PAR    ! for 5 hits

VTP2_L3_PAR       0.6           0.7
VTP2_L4_PAR       0.4           0.4
VTP2_L5_PAR       0.6           0.6

DEFINE VTP2_PR_PAR  '2F 2I' 2             ! Tolerance for track projection
        'X tolerance'  XTOL
        'Y tolerance'  YTOL
```

```
       ´Allowed gap´  GAP
       ´Minimum hits´ MINHIT
END DEFINE
DEFINE VTP2_EX_PAR  2 LIKE VTP2_PR_PAR    ! Tolerance for track extension

VTP2_PR_PAR       1.6          0.8         12           8
VTP2_EX_PAR       0.8          0.8         12          10

DEFINE VTP2_H_CUT  R 2                    ! Chisq probability cuts (1-P)
       ´Track following´ PROBF
       ´Track extension´ PROBE
END DEFINE

VTP2_H_CUT        0.999             0.999

DEFINE VTP2_YBIN  I 2                     ! Number of slices in drift direction
       ´Number of Y bins´ NBINS
END DEFINE

VTP2_YBIN         20

#
#   *******************************************************************
#   *                                                                 *
#$$ *   patrec.d  Primary vertex first guess.                         *
#   *   The calculation is done on selected tracks using              *
#   *   simply the dip angle.                                         *
#   *   The parameters set the track selection criteria               *
#   *   and the final correction for effective magnetic field.        *
#   *                                                                 *
#   *******************************************************************
#

DEFINE VTP2_V1_PAR ´1I 3F´ 2
       ´Min. numb of hits´     MINHIT
       ´Min. radius´           MINRAD
       ´Min Y offset´          YOFF
       ´Eff. field corr´       ZFCORR
END DEFINE

VTP2_V1_PAR      30          2000.0        10.0          -15.7

DEFINE pr_v1_guess ´2F 1I´ 1
       ´Primary vertex´       z
        Error                 dz
       ´Error flag´           flag
END DEFINE

######################## end of patrec parameter file #################
```

Chapter 8: DSPACK tools

8.1 DSPACK server

DSPACK may be used in the Server/Client mode. DSPACK data are kept in shared memory and the clients communicate with the server via FIFO's.

For many applications it is convenient to use the server with a number of clients which start, perform a specific operation on the server's data and then exit.

A set of simple tools like this is provided. Each tool is a small program which may take command line arguments, connects to the server and passes control to a DSPACK library routine.

8.2 Environment for running DSPACK tools

Like all DSPACK programs - the tools can read environment variables. In Client/Server operation following variables are important:

`DSPACK_FILES`	This is a location of fifo files which are used for the Server - Client communication. This location depends on installation.
`DSPACK_SERVER`	The name of the server. Each installation contains a list of servers. The system is protected from running two servers with the same name.
`DSPACK_SECTION2`	This is only used by the DSPACK server (or by programs running in local mode. It is the size of section 2 of DSPACK - to contain constants (in words). At present this section cannot be re-sized, so this variable should be set to appropriate value to fit all constants.
`DSPACKRC`	Alternative location of *.dspackrc* file - or 'NONE'

Example of DSPACK environment variables

```
setenv DSPACK_FILES /usr/spool/sockets/dspack
setenv DSPACK_SERVER server-rz
setenv DSPACK_SECTION2 500000
```

dsserver

This command starts DSPACK server. It should run in the background.

dskill

Stops the server. If the communication with the server is broken the program hangs. CNTRL C typed from the terminal will make the program delete the lock file of the server and exit. dsserver should then be killed by unix **kill** command.

8.3 Tools

8.3.1 dsdd

```
dsdd file1 file2...
```

Reads Object Description Files into server's memory. If no arguments are specified - reads from standard input.

8.3.2 dsopen

```
dsopen file [-o [-m]] [-unit n]
```

`file`	DSPACK data file name - the name is expanded using client's environment before being passed to the server.
`-o`	If present - open the file for output
`-unit n`	DSPACK I/O unit number (from 0 to 9) - default is 1
`-m`	(Only for output) - Mark all objects.

Instructs the server to open data file and read the header. If opening input file, the list of objects found is printed.

8.3.3 fsmark

```
fsmark name [-unit n]
```

`name`	Object name or regular expression
`-unit n`	DSPACK I/O unit number (from 0 to 9) - default is 1

Marks selected objects for output on given unit. If the name matches a known object name, this object is marked, otherwise name is treated as a regular expression and all matching objects are marked.

8.3.4 dsread

```
dsread [-unit n]
```

`-unit n`	DSPACK I/O unit number (from 0 to 9) - default is 1

Reads one record from a data file.

8.3.5 dsclear

```
dsclear
```

Clears section 1 of DSPACK.

8.3.6 dsclose

```
dsclose [-unit n] [-o]
```

`-unit n` DSPACK I/O unit number (from 0 to 9) - default is 1
`-o` close an output file (default is input file)

Closes DSPACK data file.

8.3.7 dswrite

```
dswrite [-unit n] [-section k]
```

`-unit n` DSPACK I/O unit number (from 0 to 9) - default is 1
`-section k`DSPACK section number (1-variables, 2-constants) - default is 1

Writes all marked objects of selected section into a data file. The object definition header is written if necessary.

8.3.8 dseval

```
dseval name
```

Evaluates DSPACK **name** and writes the result to standard output. The name must be specific - pointing to a single item (of any type). The result is written to standard output without new line. The command is designed to be used in shell scripts.

8.3.9 dsdb

```
dsdb
```

DSPACK interactive debugger. The program connects to the server and starts an interactive session, allowing the user to examine DSPACK objects. Following commands are currently available:

```
name
```

Shows object definition

```
*name
```

Shows object contents. The name may contain indexes or ranges of indexes.

Examples

```
*name - shows all entries of object  name
*name(15) - shows entry 15 of object  name
*name(1:5) - shows entries 1 to 5 of object  name
*name.item - shows element  item of object  name - all entries
*name(3:5).item - shows element  item of object  name - entries 3 to 5.
```

errors—noerrors

Turn on and off error messages.

pointers—indexes

Instructs dsdb to show pointers or indexes in subsequent commands.

drivers

Show the list of installed CLASS drivers.

msg `message [object-name]`

Sends a message to an object. If object name is absent - broadcasts the message to all installed drivers.

read `file-name`

Reads an object description file.

echo

Toggles **echo** flag for subsequent **read** commands - if **echo** is on the file is displayed while reading.

f—c—none `[-D]`

Control display of object definitions.

`f`	Show FOTRAN structures
`c`	Show C structures
`none`	Show only DSPACK description
`-D`	Together with **f** or **c** - do not show DSPACK descriptions.

printenv

Show values of local environment variables. Only variables set via *.dspackrc* file or command line arguments are shown.

?

Display a short list of available commands.

8.3.10 fdsdef

A tool which converts C header files into DSPACK object description files.
Manual in preparation

8.3.11 dss

KUIP Motif interface to DSPACK objects.
Manual in preparation

Chapter 9: Relations between objects

9.1 Pointers

Pointers are handled by DSPACK and are correctly converted by the I/O system. Pointers are are only allowed as elements of structures - the item name must start from a '*' and the item description must be the name of the object to which the pointer should point. Only pointers to valid DSPACK objects are allowed. Pointers to elements of objects are illegal (the I/O system will make them point to the beginning of the structure anyway). Pointers are not followed by the I/O system - it is the user's responsibility to save all objects needed. (however - see 'user links').

One consequence of the above limitations (imposed by the I/O system) is that the pointers may be converted at will to indexes (entry numbers). The user callable routine

```
ioldflag = ids_pnt_conv (newflag)
```

instructs DSPACK to convert pointers in all subsequent 'GET' calls into indexes if newflag=1 and to convert indexes to pointers if newflag=2. The routine returns the status of the 'flag' before the call. The conversion is performed following all pointers - this means that the 'GET' call will return an object with valid pointers or indexes and all structures below have valid pointers/indexes. If the object contains pointers already the conversion routine leaves them as they are - the same is true if conversion to indexes is attempted on an object already converted. As DSPACK objects are contiguous in memory, a pointer returned by DSPACK may be treated as a pointer to an array of structures and the index is just the array index. On the other hand if the structures are defined in a 'tree' with one object containing pointers to everything which is needed - then getting the pointer to this single object is the only DSPACK call required in order to get access to all other objects.

9.2 User's Links

As mentioned above the pointers in data are not followed by DSPACK I/O. It is sometimes convenient to join objects together and to make sure that the connections are recognized and preserved by the I/O system. This is acomplished by use of User's Links.

Each object may contain a number of links to other objects. Each link has a link-name and the target object name. All links should be specified in a single call:

```
call dsdefl (name,ivdsn,list-of-links,ierr)
```

The **list-of-links** is a character string containing pairs of link-name and target- name separated by commas. Once the link has been defined it is followed by I/O. Also the user may refer to the target object using 'link' syntax: 'name:link-name' - this is always translated into target-name.

Chapter 10: Object Resources

Definitions of all objects known to DSPACK are stored in the permanent region of DSPACK's memory. The definitions cannot be changed although new objects may be defined at any time.

There are many situations when it necessary to store some additional information about the object and associate it with object's definition. (For instance we may have to keep additional information for objects which originate in a different data handling system, some object classes may require extra information...)

DSPACK provides tha user with ability of adding arbitrary data to object definition. Each object has a link to a link-list of **resources**. Each resource is a block of user defined data (although DSPACK also uses this mechanism internally). Resources are handled by the I/O system automatically. Each resource is identified by a 4 character tag **(type)**. DSPACK uses **upper case** tags for it's own resources, it is recommended that the users use **lower case** to avoid conflicts. Each resource type may appear more than once in the resource list and the mechanism is provided to extract all resources of given type attached to given object.

10.1 Adding resource data

`pnt = ` **ids_add_res** `(name,ivdsn,resid,nwords,data,ierr)`

`name`	Character - Object name
`ivdsn`	Integer - Object handle (if 0, correct handle is returned)
`resid`	Character - Resource tag - 4 characters
`nwords`	Integer - resource length in words. If this number is negative then the memory is allocated for ABS(nwords) but nothing is copied
`data`	Data to be copied (if nwords>0)
`ierr`	Returned error flag - should be 0

ids_add_res adds a resource of type resid to object name. The pointer to the new resource is returned. If nwords¿0 then nwords words are copied from data, if nwords<0 then nothing is copied but the memory is allocated and the pointer is returned.

10.2 Getting resource data

`pnt = ` **ids_get_res** `(name,ivdsn,resid,ncnt,nwords,ierr)`

`name`	Character - Object name
`ivdsn`	Integer - Object handle (if 0, correct handle is returned)
`resid`	Character - Resource tag - 4 characters. More than one tag may be specified, separated by commas - it is interpretted as an **OR**
`ncnt`	Integer - resource counter. It allows the user to loop in order to get all resources of given type (or a list of types) - the counter should be set to 0 before the first call and then not modified in the subsequent calls. If no more resources of selected type are available the NULL pointer is returned.
`nwords`	Integer - resource length in words.
`ierr`	Returned error flag - should be 0

ids_get_res returns the pointer to a resource of given type. The call may be repeated in order to get all resources of given type.

10.3 Replacing resources

A resource block may be easily modified if the modification does not change it's length - one can simply get the pointer to a resource and overwrite the contents.

If the length of the resource has to be changed, then ids_get_res should be called to get the pointer to the resource and this is followed by a call to ids_rpl_res.

```
pnt  =  ids_rpl_res  (pnt,nwords,data,ierr)
```

`pnt`	pointer to existing resource
`nwords`	Integer - new resource length in words. If this number is negative then the memory is allocated for ABS(nwords) but nothing is copied
`data`	Data to be copied (if nwords>0)
`ierr`	Returned error flag - should be 0

ids_rpl_res returns the new pointer to a resource. If nwords¿0 then nwords words are copied from data, if nwords<0 then nothing is copied but the memory is allocated and the pointer is returned.

Access to resources - an example

```
      subroutine ds_setcl(name,ivdsn,class,itype,ierr)
*
*   *******************************************************************
*   *                                                                 *
*$$ *   subroutine ds_setcl  Set object class and type.               *
*   *   Object must be defined already.                               *
*   *   Input:                                                        *
*   *         name - character - object name                          *
*   *         ivdsn - integer - object handle                         *
*   *         class - character - class ident (may be with parentage) *
*   *         itype - integer - object type (lt 32000)                *
*   *         ierr - integer - error code:                            *
*   *            0 - OK                                               *
*   *   Output:                                                       *
*   *         ivdsn - integer - correct handle                        *
*   *                                                                 *
*   *   Ryszard Zybert    Apr 13 15:13:57 1994                        *
*   *******************************************************************
*
      character*(*) name
      character*(*) class
      character*16 cls(10)

      structure /cl_typ/
        union
          map
            integer*2 class
            integer*2 type
          end map
          map
            integer cl_type
          end map
        end union
```

```
      end structure
      record /cl_typ/ ct
      structure /cl_res/
        integer class
        character*16 name
      end structure
      record /cl_res/ cl
      pointer ( cl_p,cl )

....
....
....
                                                     ! Now deal with ancestry

      call dsfldc(class,cls,n,´.´)                   ! parse class into cls array
      if( n.gt.0 ) then
        isk = 0
        cl_p = ids_get_res(name,ivdsn,´PRTG´,isk,nw,ierr)   ! is there parentage resource?
        if( cl_p.eq.0 ) then
          cl_p = ids_add_res(name,ivdsn,´PRTG´,-n*5,0,ierr) ! if not - make one
        else
          cl_p = ids_rpl_res(cl_p,-n*5,0,ierr)              ! if yes - replace it
        endif
        if( cl_p.eq.0 ) go to 990
        do i=n,1,-1                                         ! write ancestry into resource
          cl.class = ids_nclass(cls(i))                     ! convert class to a number
          cl.name  = cls(i)                                 ! store the class name
          cl_p = cl_p + 20
        enddo
      endif

990   continue
      call ds_error(´DS_SETCL´,name,ierr,idum)

999   end
```

Chapter 11: An Example

Following is a working example using some of the features described in this guide.

User's program **patrec_ds** is a track finding code. The space points have been stored in a DSPACK data file. The user wants to process a number of events by reading this file - the resulting tracks should be written into a new data file.

The program requires **patrec.d** Object Description File shown on page 16. The file is read-in once before the event loop starts. **dsopen** and **dsread** are used to set-up the input and output data file. **fsmark** tags all required objects for output.

Additional control parameters for **patrec_ds** are kept in `$PATREC_HOME/.dspackrc` file. patrec_ds may call XGKS to produce graphics (although it does not in this example) - therefore `$PATREC_HOME/.dspackrc` file is set to be also used by XGKS as it's Xdefaults file.

patrec_ds .dspackrc file

```
:
:                                Make sure that dseval doesn´t waste time on this
:
dspack.eval.end         :
:
:                                Tell patrec that there it should not read data
:
dspack.patrec.setenv    : DSPACK_INPUT = NONE
:
:                     Make this file to be used instead of .Xdefaults by X tools
:
dspack.*.setenv         : XENVIRONMENT = .dspackrc
:
:                                Set DSPACK server environment for everybody
:
dspack.*.setenv         : DSPACK_FILES = /usr/spool/sockets/dspack
dspack.*.setenv         : DSPACK_SERVER = server-rz

:
:                                Tell patrec to process one event only
:
dspack.patrec.setenv    : EVENTS       = 1
dspack.patrec.setenv    : SKIP         = 0
:
: This is for XGKS - it will read this file because $XENVIRONMENT says so
:                    Patrec tells XGKS to use name ´patrec´ for X widgets.
:
patrec.geometry         : 850x700+10+10
patrec.Background       : DarkGreen
```

patrec_ds produces an object **rtrack_vt2_t**. In the example below the EXEC driver is programmed to call **patrec_ds** and **rtrack_vt2_t** is set to have appropriate class and type. **point_vt2_t** is an object needed by **patrec_ds** and kept in the input file. It's class is set so that **dsinput** is called when the request for **point_vt2_t** is made.

Therefore the event loop only needs to call **dswrite** - it will try to write **rtrack_vt2_t** - this will call **patrec_ds** which requires **point_vt2_t** - so **dsinput** is called. The event memory is cleared after each event with **dsclear**.

dseval is used for monitoring (and demonstration) purposes. **patrec_ds** calculates a first guess of the primary vertex position. This number with it's error is printed for each event. Additionally a new object **v1** is defined here which is programmed to calculate the mean value and error of vertex position over the number of events. The **TYPE** of this object is set to 2, so that it is re-calculated every time it is accessed - the object is stored in section 2 - so that it survives between events. It is accessed exactly once per event - by **dseval** in the event loop. After the event loop it's TYPE is changed to 0 to enable us to use it without re-calculation (as you can see from the expression - each re-calculation changes the value of the object in this case).

It is assummed that the DSPACK server is already running.

Example of using EXEC class

```
#!/bin/csh
#
#   ******************************************************************
#   *                                                                *
#$$ *   patrec.sh  C shell script to run patrec                      *
#   *   This is a simple example of using DSPACK tools with          *
#   *   Object Oriented approach.                                    *
#   *   The event loop contains only DSWRITE and DSCLEAR,            *
#   *   PATREC as well as DSREAD are triggered by FILL messages      *
#   *   which are sent to objects.                                   *
#   *   All this is done using a single CLASS - EXEC                 *
#   *                                                                *
#   *   Ryszard Zybert    Jun 17 20:26:46 1994                       *
#   ******************************************************************
#
#
#   ******************************************************************
#   *                                                                *
#$$ *   patrec.sh  Tools used:                                       *
#   *                                                                *
#   *   dsdd - read object description files                         *
#   *   dsopen - open DSPACK data files (input and output)           *
#   *   dsread - read one record from input file                     *
#   *   dswrite - write one event to output file                     *
#   *   dseval - evaluate and return the value of a DSPACK object    *
#   *   fsmark - mark an object for output                           *
#   *   dsclear - clear event data (section 1 of DSPACK)             *
#   *   dsclose - close data file                                    *
#   *                                                                *
#   *   patrec_ds is the tracking program:                           *
#   *             uses point_vt2_t                                   *
#   *             produces rtrack_vt2_t                              *
#   *                                                                *
#   *   Both patrec_ds and dsread are called automatically their     *
#   *   objects are requested                                        *
#   *                                                                *
#   ******************************************************************
#
set nevnts = $1
set fname_in   = $2
set fname_out  = $3

#
```

```
# Read object definition file and PATREC parameters
#
dsdd $PATREC_HOME/par/patrec.d

#
# Open data file and read the parameter block
#
dsopen $fname_in
dsread

#
# Define more objects and classes
#
dsdd <<.end

INIT .exec(1)
     methods(1).command  = dsread >/dev/null
     methods(1).messages = FILL
END INIT

INIT .exec(2)
     methods(1).command  = $PATREC_HOME/bin/patrec_ds >/dev/null
     methods(1).messages = FILL
END INIT

DEFINE tracks I 1
   number_of_tracks n
END DEFINE

CALC tracks.n = NENT(rtrack_vt2_t)

RECLASS rtrack_vt2_t -CLASS exec  -TYPE 2
RECLASS point_vt2_t  -CLASS exec  -TYPE 1


#
# Define extra calculator stuff to get mean vertex position
# Note - -TYPE 2 for calculator objects means they are recalculated
# every time they are accessed.... careful!!
#

DEFINE v1 S 2
        .REAL4  sum
        .REAL4  ss
        .INT4   n
        .REAL4  z
        .REAL4  dz
END DEFINE

CALC v1.sum = v1.sum + pr_v1_guess.z
CALC v1.ss  = v1.ss  + pr_v1_guess.dz^2
CALC v1.n   = v1.n   + 1
CALC v1.z   = v1.sum/v1.n
CALC v1.dz  = SQRT(v1.ss)/v1.n

RECLASS v1 -CLASS calc -TYPE 2
```

```
INIT v1
   sum = 0.0
   ss  = 0.0
   n   = 0
END INIT

.end

#
# Open output file
#
dsopen $fname_out -o

#
# Mark objects for output
#
fsmark rtrack_vt2_t
fsmark event_mc_t
fsmark vertex_mc_t
fsmark track_mc_t
fsmark ref_vt2_mc_t
fsmark event_t
fsmark point_vt2_t
fsmark vt2_t

setenv DSPACKRC $PATREC_HOME/.dspackrc
setenv DSPACK_MODE 1

#
# Loop over events
#
# dseval calls don´t take part in the processing - they are here
# simply to echo values of some objects for monitoring
# It should not be over-done - it keeps server busy.
# However - evaluating v1.z is important - otherwise the sums whould
# not be calculated.
#
echo "Starting event loop"

set n = 0
while ( $n < $nevnts )
  dsclear
  dswrite >/dev/null
  @ n++

  echo "Event" `dseval ´event_t.n_event´`": " `dseval ´tracks.n´` "tracks, " \
       "Vertex: " `dseval ´pr_v1_guess.z´`" +/-"\
       `dseval ´pr_v1_guess.dz´`", mean: "\
       `dseval ´v1.z´`
end

#
# Close data files
#
dsclose
dsclose -o
```

```
#
# Have a look at mean vertex - make sure it does not recalculate!!
# Setting -TYPE 0 for v1 stops it from responding to REFILL message.
#
dsdd <<.end
RECLASS v1 -CLASS calc -TYPE 0
.end

echo "Mean vertex over "`dseval ´v1.n´`" events: "\
                        `dseval ´v1.z´`" +/- "`dseval ´v1.dz´`
```

Here is an example interactive session using the shell script above:

Interactive session with DSPACK tools

```
np_rhi/data/tmp> dsserver&
[1] 964
np_rhi/data/tmp> patrec.sh 10 e50t50.ds p_e10t50.ds
                                                         (dsdd starts)
*****************************************************
*   This is $HOME/.dspackrc file                    *
*****************************************************
DSPACK 1.128, 22 Jun 1994
######################## patrec parameter file #####################
                                                         (long listing removed)
######################## end of patrec parameter file ##############

                                                         (dsopen starts)
*****************************************************
*   This is $HOME/.dspackrc file                    *
*****************************************************
DSPACK 1.128, 22 Jun 1994
Read definitions
/*
* DSPACK c header file generated from a data file
*
* DSPACK file created on 29 May 1994 at 19:46
* By DSPACK 1.077, 25 May 1994
* Following objects found on input    1
*   .api_def          .event_mc_t       event_mc_t        .vertex_mc_t
*   vertex_mc_t       .track_mc_t       track_mc_t        .ref_vt2_mc_t
*   ref_vt2_mc_t      .event_t          event_t           .vt2_t
*   point_vt2_t       .point_vt2_t      rtrack_vt2_t      .rtrack_vt2_t
*   vt2_t
*
* Following definitions may be used as a header
* to define all necessary structures
*/
typedef struct api_def
{
                char api_def_pack[52]
};
typedef struct event_mc_t
{
                char packed[1172]
```

```
};
typedef struct vertex_mc_t
{
                struct vertex_sim_mc_t *vertex_sim_p;
                struct vertex_mc_t *next_vertex_p;
                struct track_mc_t *tdaughter_p;
                struct track_mc_t *tparent_p;
                int n_daughter;
                int l_vertex;
                float z;
                float y;
                float x;
};
typedef struct ref_vt2_mc_t
{
                float pz;
                float py;
                float px;
                float z;
                float y;
                float x;
                int n_row;
                int n_sec;
};
typedef struct track_mc_t
{
                struct track_mc_t *next_track_p;
                struct track_sim_mc_t *track_sim_p;
                struct track_mc_t *next_vtrack_p;
                struct track_mc_t *next_daughter_p;
                struct track_mc_t *tdaughter_p;
                struct track_mc_t *tur_p;
                struct track_mc_t *tparent_p;
                struct vertex_mc_t *vstop_p;
                struct vertex_mc_t *vstart_p;
                struct ref_mtr_mc_t *ref_mtr_p;
                struct ref_mtl_mc_t *ref_mtl_p;
                struct ref_vt2_mc_t *ref_vt2_p;
                struct ref_vt1_mc_t *ref_vt1_p;
                int l_track;
                float charge;
                int id;
                float ptot;
                float pt;
                float y;
                float pz;
                float py;
                float px;
};
typedef struct event_mc_t
{
                struct event_sim_mc_t *event_sim_p;
                struct track_mc_t *track_p;
                struct vertex_mc_t *vertex_p;
                int n_track_prim;
                int n_vertex;
                int l_vertex;
```

```
                float eg_version;
                int time;
                int date;
                int n_targ_spec_prot;
                int n_targ_spec_neut;
                int n_proj_spec_prot;
                int n_proj_spec_neut;
                float b;
                int m_trig;
                float e_lab;
                float target_z;
                float target_a;
                float proj_z;
                float proj_a;
                int n_event;
                int n_run;
                char eg_name[32]
};
typedef struct vertex_mc_t
{
                char packed[500]
};
typedef struct track_mc_t
{
                char packed[1304]
};
typedef struct ref_vt2_mc_t
{
                char packed[352]
};
typedef struct event_t
{
                char packed[1076]
};
typedef struct point_vt2_t
{
                struct point_vt2_t *next_trkpoint_p;
                struct point_vt2_t *next_point_p;
                struct track_mc_t *track_mc_p;
                int iflag;
                int max_adc;
                float charge;
                float var_y;
                float var_x;
                float covar[6];
                float z;
                float y;
                float x;
                int n_row;
                int n_sec;
};
typedef struct rtrack_vt2_t
{
                struct rtrack_vt2_t *next_segment_p;
                struct rtrack_vt2_t *next_rtrack_p;
                struct track_vt2_t *track_p;
                struct point_vt2_t *point_p;
```

```
                struct track_mc_t *track_mc_p;
                int iflag;
                float var_tmean;
                float tmean_charge;
                float pchi2;
                int n_point;
                float z_last;
                float y_last;
                float x_last;
                float z_first;
                float y_first;
                float x_first;
                float covar[15];
                float phi;
                float tanl;
                float qpxz;
                float z;
                float y;
                float x;
};
typedef struct vt2_t
{
                struct svertex_vt2_t *svertex_p;
                struct mvertex_vt2_t *mvertex_p;
                struct rtrack_vt2_t *rtrack_p;
                struct point_vt2_t *point_p;
                struct slow_vt2_t *slowcontrol_p;
                float clock_to_trigger;
                float tslice;
                float delay;
};
typedef struct event_t
{
                struct event_mc_t *event_mc_p;
                struct dss_t *dss_p;
                struct beam_t *beam_p;
                struct ring_t *ring_p;
                struct veto_t *veto_p;
                struct tofr_t *tofr_p;
                struct tofl_t *tofl_p;
                struct mtr_t *mtr_p;
                struct mtl_t *mtl_p;
                struct vt2_t *vt2_p;
                struct vt1_t *vt1_p;
                struct bos_t *bos_p;
                struct rundb_t *rundb_p;
                int time;
                int date;
                int mc_flag;
                int mask_trig;
                float target_mass;
                float proj_mass;
                float momentum;
                int n_event;
                int n_run;
};
typedef struct vt2_t
```

```
{
                char packed[500]
};
typedef struct point_vt2_t
{
                char packed[640]
};
typedef struct rtrack_vt2_t
{
                char packed[1012]
};

                                                            (dsread starts)
*****************************************************
*   This is $HOME/.dspackrc file                    *
*****************************************************
DSPACK 1.128, 22 Jun 1994
Read record type:   2

                                                            (dsdd starts)
*****************************************************
*   This is $HOME/.dspackrc file                    *
*****************************************************
DSPACK 1.128, 22 Jun 1994

INIT .exec(1)
     methods(1).command  = dsread >/dev/null
     methods(1).messages = FILL
END INIT

INIT .exec(2)
     methods(1).command  = /data/tmp/patrec/bin/patrec_ds >/dev/null
     methods(1).messages = FILL
END INIT

DEFINE tracks I 1
   number_of_tracks n
END DEFINE

CALC tracks.n = NENT(rtrack_vt2_t)

RECLASS rtrack_vt2_t -CLASS exec  -TYPE 2
RECLASS point_vt2_t  -CLASS exec  -TYPE 1


#
# Define extra calculator stuff to get mean vertex position
# Note - -TYPE 2 for calculator objects means they are recalculated
# every time they are accessed.... careful!!
#

DEFINE v1 S 2
        .REAL4  sum
        .REAL4  ss
        .INT4   n
        .REAL4  z
        .REAL4  dz
```

```
END DEFINE

CALC v1.sum = v1.sum + pr_v1_guess.z
CALC v1.ss  = v1.ss  + pr_v1_guess.dz^2
CALC v1.n   = v1.n   + 1
CALC v1.z   = v1.sum/v1.n
CALC v1.dz  = SQRT(v1.ss)/v1.n

RECLASS v1 -CLASS calc -TYPE 2

INIT v1
   sum = 0.0
   ss  = 0.0
   n   = 0
END INIT

                                                         (dsopen -o starts)
****************************************************
*   This is $HOME/.dspackrc file                   *
****************************************************
DSPACK 1.128, 22 Jun 1994

                                                         (fsmark)
rtrack_vt2_t     marked for output 1
event_mc_t       marked for output 1
vertex_mc_t      marked for output 1
track_mc_t       marked for output 1
ref_vt2_mc_t     marked for output 1
event_t          marked for output 1
point_vt2_t      marked for output 1
vt2_t            marked for output 1


Starting event loop
Event 1:  52 tracks,  Vertex:  -542.457 +/- 9.76601, mean:  -542.457
Event 2:  54 tracks,  Vertex:  -539.575 +/- 11.1741, mean:  -541.016
Event 3:  54 tracks,  Vertex:  -542.869 +/- 11.7035, mean:  -541.633
Event 4:  52 tracks,  Vertex:  -542.335 +/- 11.6563, mean:  -541.809
Event 5:  51 tracks,  Vertex:  -531.569 +/- 14.2423, mean:  -539.761
Event 6:  51 tracks,  Vertex:  -540.852 +/- 8.69483, mean:  -539.943
Event 7:  53 tracks,  Vertex:  -539.459 +/- 13.6912, mean:  -539.874
Event 8:  54 tracks,  Vertex:  -535.783 +/- 14.0949, mean:  -539.362
Event 9:  50 tracks,  Vertex:  -539.109 +/- 13.6300, mean:  -539.334
Event 10:  53 tracks,  Vertex:  -543.064 +/- 14.7827, mean:  -539.707

DSPACK 1.128, 22 Jun 1994                                (dsclose)
DSPACK 1.128, 22 Jun 1994                                (dsclose -o)
DSPACK 1.128, 22 Jun 1994                                (dsdd)
RECLASS v1 -CLASS calc -TYPE 0
Mean vertex over 10 events:  -539.707 +/- 3.95214


np_rhi/data/tmp> dsdb
            *** dsdb - DSPACK interactive debugger ***
DSPACK 1.128, 22 Jun 1994
dsdb> ls -l
Name             Type        Id     Sect    Size    Items   Nent    Bytes
```

```
POINT              R         18        0        3        3        0         0
vt2_hit            S         19        1        6        4     2902     69648
vt2_hitp           I         20        1        1        1     2902     11608
VTP2_SECTORS       S         21        2       11       11        1        44
VTP2_DIMS          S         22        2        5        5        1        20
VTP2_L3_PAR        R         23        2        2        2        1         8
VTP2_L4_PAR        R         24        2        2        2        1         8
VTP2_L5_PAR        R         25        2        2        2        1         8
VTP2_PR_PAR        M         26        2        4        4        1        16
VTP2_EX_PAR        M         27        2        4        4        1        16
VTP2_H_CUT         R         28        2        2        2        1         8
VTP2_YBIN          I         29        2        1        1        1         4
VTP2_V1_PAR        M         30        2        4        4        1        16
pr_v1_guess        M         31        1        3        3        1        12
event_mc_t         S         45        1       30       23        1       120
vertex_mc_t        S         47        1        9        9        1        36
track_mc_t         S         49        1       22       22       50      4400
ref_vt2_mc_t       S         51        1        8        8       50      1600
event_t            S         53        1       22       22        1        88
point_vt2_t        S         55        1       19       14     2902    220552
rtrack_vt2_t       S         57        1       37       23       53      7844
vt2_t              S         59        1        8        8        1        32
tracks             I         60        1        1        1        1         4
v1                 S         61        2        5        5        1        20
helix              M         72        0        9        9        0         0
pr_track           S         73        1       12        4       53      2544
pr_hits            S         74        1       72        1       53     15264
pr_time            R         75        1        1        1        1         4
pr_res             S         76        1      144        2       53     30528

dsdb>  rtrack_vt2_t
 Definition of data set: rtrack_vt2_t        (rtra) ID:    57
 Type:                   Structure
 Section:                    1
 Machine words:             37
 Items:                     23
   1  *next_segment_p   rtrack_vt2_t
   2  *next_rtrack_p    rtrack_vt2_t
   3  *track_p          track_vt2_t
   4  *point_p          point_vt2_t
   5  *track_mc_p       track_mc_t
   6  iflag             .INT4
   7  var_tmean         .REAL4
   8  tmean_charge      .REAL4
   9  pchi2             .REAL4
  10  n_point           .INT4
  11  z_last            .REAL4
  12  y_last            .REAL4
  13  x_last            .REAL4
  14  z_first           .REAL4
  15  y_first           .REAL4
  16  x_first           .REAL4
  17  covar(15)         .REAL4
  18  phi               .REAL4
  19  tanl              .REAL4
  20  qpxz              .REAL4
  21  z                 .REAL4
```

```
  22  y                .REAL4
  23  x                .REAL4

typedef struct vertex_mc_t
{
                struct vertex_sim_mc_t *vertex_sim_p;
                struct vertex_mc_t *next_vertex_p;
                struct track_mc_t *tdaughter_p;
                struct track_mc_t *tparent_p;
                int n_daughter;
                int l_vertex;
                float z;
                float y;
                float x;
};
typedef struct ref_vt2_mc_t
{
                float pz;
                float py;
                float px;
                float z;
                float y;
                float x;
                int n_row;
                int n_sec;
};
typedef struct track_mc_t
{
                struct track_mc_t *next_track_p;
                struct track_sim_mc_t *track_sim_p;
                struct track_mc_t *next_vtrack_p;
                struct track_mc_t *next_daughter_p;
                struct track_mc_t *tdaughter_p;
                struct track_mc_t *tur_p;
                struct track_mc_t *tparent_p;
                struct vertex_mc_t *vstop_p;
                struct vertex_mc_t *vstart_p;
                struct ref_mtr_mc_t *ref_mtr_p;
                struct ref_mtl_mc_t *ref_mtl_p;
                struct ref_vt2_mc_t *ref_vt2_p;
                struct ref_vt1_mc_t *ref_vt1_p;
                int l_track;
                float charge;
                int id;
                float ptot;
                float pt;
                float y;
                float pz;
                float py;
                float px;
};
typedef struct point_vt2_t
{
                struct point_vt2_t *next_trkpoint_p;
                struct point_vt2_t *next_point_p;
                struct track_mc_t *track_mc_p;
                int iflag;
```

```
                int max_adc;
                float charge;
                float var_y;
                float var_x;
                float covar[6];
                float z;
                float y;
                float x;
                int n_row;
                int n_sec;
};
typedef struct rtrack_vt2_t
{
                struct rtrack_vt2_t *next_segment_p;
                struct rtrack_vt2_t *next_rtrack_p;
                struct track_vt2_t *track_p;
                struct point_vt2_t *point_p;
                struct track_mc_t *track_mc_p;
                int iflag;
                float var_tmean;
                float tmean_charge;
                float pchi2;
                int n_point;
                float z_last;
                float y_last;
                float x_last;
                float z_first;
                float y_first;
                float x_first;
                float covar[15];
                float phi;
                float tanl;
                float qpxz;
                float z;
                float y;
                float x;
};
dsdb>  *rtrack_vt2_t(10:11)
 Data set: rtrack_vt2_t(10:11)
 Entry:    10
 Type:                  Structure   (at: -287403752)
   1  *next_segment_p  Pointer        : 0
   2  *next_rtrack_p   Pointer        : -287403604
   3  *track_p         Pointer        : 0
   4  *point_p         Pointer        : -287653192
   5  *track_mc_p      Pointer        : 0
   6  iflag            Integer        : 0
   7  var_tmean        Real           : 0.
   8  tmean_charge     Real           : 0.
   9  pchi2            Real           : 0.
  10  n_point          Integer        : 46
  11  z_last           Real           : 111.000
  12  y_last           Real           : -6.42164
  13  x_last           Real           : 70.4347
  14  z_first          Real           : -28.0000
  15  y_first          Real           : -5.00439
  16  x_first          Real           : 13.2499
```

```
       17  covar(1)        Real             : 0.
       17  covar(2)        Real             : 0.
       17  covar(3)        Real             : 0.
       17  covar(4)        Real             : 0.
       17  covar(5)        Real             : 0.
       17  covar(6)        Real             : 0.
       17  covar(7)        Real             : 0.
       17  covar(8)        Real             : 0.
       17  covar(9)        Real             : 0.
       17  covar(10)       Real             : 0.
       17  covar(11)       Real             : 0.
       17  covar(12)       Real             : 0.
       17  covar(13)       Real             : 0.
       17  covar(14)       Real             : 0.
       17  covar(15)       Real             : 0.
  18  phi               Real           : 0.
  19  tanl              Real           : 0.
  20  qpxz              Real           : 0.
  21  z                 Real           : 0.
  22  y                 Real           : 0.
  23  x                 Real           : 0.
Entry:    11
Type:                   Structure   (at: -287403604)
   1  *next_segment_p   Pointer        : 0
   2  *next_rtrack_p    Pointer        : -287403456
   3  *track_p          Pointer        : 0
   4  *point_p          Pointer        : -287595584
   5  *track_mc_p       Pointer        : 0
   6  iflag             Integer        : 0
   7  var_tmean         Real           : 0.
   8  tmean_charge      Real           : 0.
   9  pchi2             Real           : 0.
  10  n_point           Integer        : 24
  11  z_last            Real           : 111.000
  12  y_last            Real           : -2.66645
  13  x_last            Real           : 31.4202
  14  z_first           Real           : -111.000
  15  y_first           Real           : -1.78658
  16  x_first           Real           : 16.8726
       17  covar(1)        Real             : 0.
       17  covar(2)        Real             : 0.
       17  covar(3)        Real             : 0.
       17  covar(4)        Real             : 0.
       17  covar(5)        Real             : 0.
       17  covar(6)        Real             : 0.
       17  covar(7)        Real             : 0.
       17  covar(8)        Real             : 0.
       17  covar(9)        Real             : 0.
       17  covar(10)       Real             : 0.
       17  covar(11)       Real             : 0.
       17  covar(12)       Real             : 0.
       17  covar(13)       Real             : 0.
       17  covar(14)       Real             : 0.
       17  covar(15)       Real             : 0.
  18  phi               Real           : 0.
  19  tanl              Real           : 0.
  20  qpxz              Real           : 0.
```

```
  21  z                 Real          : 0.
  22  y                 Real          : 0.
  23  x                 Real          : 0.
dsdb>  *rtrack_vt2_t.y_first
    1  rtrack_vt2_t.y_first  =       -27.60115
    2  rtrack_vt2_t.y_first  =       -14.75850
    3  rtrack_vt2_t.y_first  =       -11.03135
    4  rtrack_vt2_t.y_first  =        -9.14805
    5  rtrack_vt2_t.y_first  =        -8.13782
    6  rtrack_vt2_t.y_first  =        -7.49848
    7  rtrack_vt2_t.y_first  =        -7.28085
    8  rtrack_vt2_t.y_first  =        -6.28944
    9  rtrack_vt2_t.y_first  =        -4.25609
   10  rtrack_vt2_t.y_first  =        -5.00439
   11  rtrack_vt2_t.y_first  =        -1.78658
...                                                  (long listing removed)
   52  rtrack_vt2_t.y_first  =        -1.54501
   53  rtrack_vt2_t.y_first  =       -12.39243
dsdb>  indexes
Showing indexes
dsdb>  *rtrack_vt2_t(11)
 Data set: rtrack_vt2_t(11)
 Entry:    11
 Type:                  Structure   (at: -287403604)
   1  *next_segment_p   Index         : 0
   2  *next_rtrack_p    Index         : 12
   3  *track_p          Index         : 0
   4  *point_p          Index         : 2209
   5  *track_mc_p       Index         : 0
   6  iflag             Integer       : 0
   7  var_tmean         Real          : 0.
   8  tmean_charge      Real          : 0.
   9  pchi2             Real          : 0.
  10  n_point           Integer       : 24
  11  z_last            Real          : 111.000
  12  y_last            Real          : -2.66645
  13  x_last            Real          : 31.4202
  14  z_first           Real          : -111.000
  15  y_first           Real          : -1.78658
  16  x_first           Real          : 16.8726
         17  covar(1)        Real              : 0.
         17  covar(2)        Real              : 0.
         17  covar(3)        Real              : 0.
         17  covar(4)        Real              : 0.
         17  covar(5)        Real              : 0.
         17  covar(6)        Real              : 0.
         17  covar(7)        Real              : 0.
         17  covar(8)        Real              : 0.
         17  covar(9)        Real              : 0.
         17  covar(10)       Real              : 0.
         17  covar(11)       Real              : 0.
         17  covar(12)       Real              : 0.
         17  covar(13)       Real              : 0.
         17  covar(14)       Real              : 0.
         17  covar(15)       Real              : 0.
  18  phi               Real          : 0.
  19  tanl              Real          : 0.
```

```
  20  qpxz              Real           : 0.
  21  z                 Real           : 0.
  22  y                 Real           : 0.
  23  x                 Real           : 0.
dsdb>   *point_vt2_t(2209)
 Data set: point_vt2_t(2209)
 Entry:  2209
 Type:                  Structure   (at: -287595584)
   1  *next_trkpoint_p  Index          : 2258
   2  *next_point_p     Index          : 2210
   3  *track_mc_p       Index          : 39
   4  iflag             Integer        : 0
   5  max_adc           Integer        : 0
   6  charge            Real           : 0.
   7  var_y             Real           : 0.
   8  var_x             Real           : 0.
        9  covar(1)         Real              : 5.58987E-04
        9  covar(2)         Real              : 0.
        9  covar(3)         Real              : 6.25000E-04
        9  covar(4)         Real              : 0.
        9  covar(5)         Real              : 0.
        9  covar(6)         Real              : 0.
  10  z                 Real           : -111.000
  11  y                 Real           : -1.78658
  12  x                 Real           : 16.8726
  13  n_row             Integer        : 1
  14  n_sec             Integer        : 4
dsdb> pointers
Showing pointers
dsdb> *point_vt2_t(2209)
 Data set: point_vt2_t(2209)
 Entry:  2209
 Type:                  Structure   (at: -287595584)
   1  *next_trkpoint_p  Pointer        : -287591860
   2  *next_point_p     Pointer        : -287595508
   3  *track_mc_p       Pointer        : -287766292
   4  iflag             Integer        : 0
   5  max_adc           Integer        : 0
   6  charge            Real           : 0.
   7  var_y             Real           : 0.
   8  var_x             Real           : 0.
        9  covar(1)         Real              : 5.58987E-04
        9  covar(2)         Real              : 0.
        9  covar(3)         Real              : 6.25000E-04
        9  covar(4)         Real              : 0.
        9  covar(5)         Real              : 0.
        9  covar(6)         Real              : 0.
  10  z                 Real           : -111.000
  11  y                 Real           : -1.78658
  12  x                 Real           : 16.8726
  13  n_row             Integer        : 1
  14  n_sec             Integer        : 4
dsdb>  v1
 v1.sum = v1.sum + pr_v1_guess.z
 v1.ss  = v1.ss  + pr_v1_guess.dz^2
 v1.n   = v1.n   + 1
 v1.z   = v1.sum/v1.n
```

```
 v1.dz  = SQRT(v1.ss)/v1.n
 Definition of data set: v1                 (v1  ) ID:    61
 Type:                   Structure
 Section:                    2
 Machine words:              5
 Items:                      5
   1  sum               .REAL4
   2  ss                .REAL4
   3  n                 .INT4
   4  z                 .REAL4
   5  dz                .REAL4

typedef struct v1
{
                float sum;
                float ss;
                int n;
                float z;
                float dz;
};
dsdb>  *v1
 Data set: v1
 Entry:     1
 Type:                  Structure   (at: -279883728)
   1  sum               Real           : -5397.07
   2  ss                Real           : 1561.94
   3  n                 Integer        : 10
   4  z                 Real           : -539.707
   5  dz                Real           : 3.95214
dsdb>  quit
```