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.