DSPACK

Release Notes
(Version 1.28)

Ryszard Zybert
School of Physics and Space Research
University of Birmingham

12 May 1995

Contents

1 Overview

In the last few months more and more people started using DSPACK. This led to very many requests for new features or for changes and almost as many bug reports. Not all the requests could be fulfilled at this stage and - I am sure - not all the bugs have been removed. But the number of changes justifies - in my opinion - a major release.

New features and tools are covered by the following sections. The new version is I/O compatible with the previous one. The changes which may require user's action are as follows:

- All Fortran I/O has been removed. This solved several problems where platform dependent interference between C and Fotran I/O produced unexpected results. This means that wherever manual refers to a 'logical unit number' - from now on this will mean C file descriptor. For Fortran programmers routines are provided to open and close files and to produce text output - all using C I/O.

- All dependence on CERN library has been removed. Consequently - the HEPDB driver has been moved to libdb.a.
- For Solaris and HP-UX DSPACK is now provided as shared library. In order to use this feature your program should be linked with

  -L$(DSPACK_HOME)/lib -ldspack
- Access to all DSPACK functions from C and C++ is now via interface routines rather than cpp macros. The dsc.h header file is still there for compatibility but it is empty and not needed anymore. For ANSI C and especially for C++ a header file dsc_prot.h is provided with prototypes for all calls.
- In server/client mode shared memory handling has been modified in such a way that a failure to create shared memory segment does not stop the system - the error is reported back to the client who requested memory. In order to make use of this feature the programs much check the error condition when creating new object or resizing existing ones.

2 New Features

2.1 Simplified Calling Sequences

Practice showed that some of the arguments in original DSPACK calls were not used at all. Therefore - while preserving the old calls - new, simpler calling sequences were introduced. This is a continuation of the process which started some time ago. The new calls are listed below:

```
int ds_init(char* name, int* ierr)
subroutine ds_init(name,ierr)
character*(*) name
integer ierr
```

New initialization routine. It expects that MODE and SERVER name are set via environment variables or command line arguments - otherwise it defaults to LOCAL mode. Like all DSPACK functions - in C it returns ZERO on failure and non-zero on success.

```
int ds_clear()
subroutine ds_clear()
```

Clears SECTION 1 of DSPACK.

```
int ds_read(int* ierr)
subroutine ds_read(ierr)
```

Reads one logical record from all open inputs.

```
int ds_write(int isect, int* ierr)
subroutine ds_write(isect,ierr)
```

Writes all objects of section isect to all open output streams for which they were marked.

```
int ds_close_in()
subroutine ds_close_in()
```

Closes all open inputs

```
int ds_close_out()
subroutine ds_close_out()
```

Closes all open outputs

2.2 Object Manipulation

```
int ds_map_object(char* name1, int* iv1, char* name2,
                                    int* iv2, int* ierr)
subroutine ds_map_object(name1, iv1, name2, iv2, ierr)
character*(*) name1, name2
integer iv1,iv2,ierr
```

Temporarily maps object **name1** to memory occupied by object **name2** iv1 and iv2 are corresponding object handles. Mapping is temporary in the sense that object **name1** is not updated if object **name2** moves or is deleted and the pointer to object **name1** may become an illegal address in such case.

```
int ds_cp_object(char* name1, int* iv1, char* name2,
                            int* iv2, int isect, int* ierr)
subroutine ds_cp_object(name1, iv1, name2, iv2, isect, ierr)
character*(*) name1, name2
integer iv1,iv2,isect,ierr
```

Copy object **name1** to **name2**. If object **name1** is not defined it will be defined using **name2** as a template - except that if object **name2** contains pointers to itself - they are renamed. **isect** is the section number for the new object (if it is not defined already). If isect=0 the section of **name1** is taken.

```
int ds_mv_object(char* name1, int* iv1, char* name2,
                                       int* iv2, int* ierr)
subroutine ds_mv_object(name1, iv1, name2, iv2,  ierr)
character*(*) name1, name2
integer iv1,iv2,ierr
```

Move data from object **name1** to **name2**. After this call object **name1** has no entries. Object **name2** will be defined if necessary. Unless **name2** is defined and is in different section from **name1** - no data are moved - just pointers. If object **name2** has to be defined the pointers to itself are renamed appropriately. However - no action is taken for pointers in other objects pointing to **name1**. DSPACK's strong typing means that these pointers are now illegal and will be cleared for I/O.

```
int ds_rm_object(char* name, int* iv, int* ierr)
subroutine ds_rm_object(name, iv, ierr)
character*(*) name
integer iv,ierr
```

Deletes object - identical to **dsdrop**

```
int ids_pnt2h(void* iaddr, int* ient)
integer function ids_pnt2h(iaddr,ient)
integer iaddr,ient
```

If **iaddr** points to a DSPACK object it returns the object handle and (in ient) the entry number.

2.3 Dynamic Pointer Binding

Pointers returned by DSPACK functions refer to current state of data. While handles are guaranteed to stay constant, when data move (for instance for new event) the pointers have to be obtained again - for instance by calling **idsget_ds**. *Dynamic Pointer Binding* allows the programmer to declare a pointer to DSPACK and request this pointer to be *bound* to a named object. A pointer declared like this will always point to the beginning of data for given object - or will be NULL if the object has no data. Pointer binding works in LOCAL and SERVER/CLIENT mode - this means that if one client moves an object all other clients which bound pointers to this object will update their pointers.

Warning: In some cases pointer binding may be time consumming. This happens if the object to which the pointer is bound has a very deep pointer structure and the data are read from tape or disk. In the process of updating the bound pointer DSPACK converts all reachable pointers to a form required in memory operation. As a consequence it is not advisable to bind pointers which are not used.

```
int ds_bind(void** pnt, char* name, int* ierr)
subroutine ds_bind(pnt, name, ierr)
integer pnt
character*(*) name
integer ierr
```

Binds *pnt (pnt in Fortran) to object **name**. This function may be called at any time (even before **ds_init**) independently whether the object **name** is defined at this moment.

Example

```
#include "event_t.h"
struct event_t *event;

/*
*   *******************************************************
*   *
*$$ *   main  Example of dynamic pointer binding
*   *   event is a pointer to "event_t" and DSPACK
*   *   keeps it up to date.
*   *   Note - checks on vt2_p and point_p are needed in the
*   *   real application - skipped here for clarity.
*   *
*   *   Ryszard Zybert    May 11 11:29:54 1995
*   *******************************************************
*/
```

```
main(int argc,char **argv)
{
  int ierr;

  ds_carg(argv);                /* Pass command line arguments */

  ds_bind(&event,"event_t",&ierr);  /* Bind event to "event_t" */

  ds_init("bind-test",&ierr);       /* Start DSPACK */

  fsopin(1,"$my_input"," ",&ierr);  /* Open input file */

  while (ds_read(&ierr))            /* Read one logical record */
    {
      if( event )               /* Check if "event_t" is there */
        {
          printf("event: %i\n",event->n_event);
          printf("First point: x: %f, y: %f, z: %f\n",
                 event->vt2_p->point_p->x,
                 event->vt2_p->point_p->y,
                 event->vt2_p->point_p->z);
        }
    }
}
```

2.4 Extended Object Description Language

Several new features have been introduced. Some of the features existed in an experimental mode in the previous versions but are ready now for 'official' release.

2.4.1 SET and STATIC

Symbols may be defined in Object Description Files in the way similar to cpp *define*.

`SET <symbol> <value>`

defines symbol - stays valid in the current file and all files included from it.

`STATIC <symbol> <value>`

like SET except that the symbol stays defined for the lifetime of the program.

2.4.2 Pre-defined Symbols

A number of symbols have been pre-defined to form a set of keywords which may be used in all Object Description Files.

Types: int,float,real,double,char4,char16,char80,char256,logical,unsigned

DSPACK sections: DS_TEMPLATE, DS_VARIABLE, DS_CONSTANT

Object type: STRUCTURE

2.4.3 Functions

A number of functions has been introduced. The syntax is function([arg]). Each function is evaluated while reading the file and it's value is substituted directly into the text read.

FILE() - full name of the file being read

TOP_ENTRY(¡name¿) - maximum existing entry number of an object

NEXT_ENTRY(¡name¿) - maximum entry number + 1

A shortcut for NEXT_ENTRY(¡name¿) may be used - if the entry number is specified as (*) - new entry is added and used.

Example:

```
# add an entry to object names and store the current file name
INIT names(*)
      file = FILE()
END INIT

# set n_files.n_read to number of files read in

SET N TOP_ENTRY(names)

INIT n_files
      n_read = N
END INIT
```

2.4.4 typedef and struct

Syntax of C *typedef* statement contains almost all information needed to define DSPACK structure. The accepted syntax has been extended in order to make use of this. In addition to what is needed by C, DSPACK requires that the section (DS_TEMPLATE, DS_CONSTANT, DS_VARIABLE) is specified. In order for C to be able to read the same header file these keywords have to be defined so that *cpp* would remove them from the text.

Example:

```
#define DS_TEMPLATE
#define DS_CONSTANT
#define DS_VARIABLE

typedef struct point DS_TEMPLATE { /* Generic point definition */
   float x;                        /* Position */
```

```
   float y;
   float z;
   struct track *track;            /* Pointer to track */
} point;

# point is just a TYPE, we now define a structure my_point
# in DS_VARIABLE section

struct point DS_VARIABLE my_point;
```

2.4.5 .read_once object

In some situations the program may attempt to read the same file several times. A typical example is when running event loop in a 'start-stop' mode where each program is started and stopped once per event. Each program may test whether the appropriate Object Description Files has been read in but an easier way may be to define a special object **.read_once**:

```
DEFINE .read_once STRUCTURE CONSTANT
      char256 file
END DEFINE
```

Once defined, this object will collect names of all files read in and will cause the repeated read requests for the same file to be ignored.

2.4.6 Environment Variables Substitution

Envinronment variables may be used to initialize DSPACK object with the syntax identical to *make*:

```
${<name>}
```

Example:

```
INIT my_name
      name = ${USER}
END INIT
```

2.4.7 Using DSPACK objects in initialization

Data stored in existing DSPACK objects may be used to initialize other objects. The fully qualified name must be enclosed in braces to be substituted by the text representing the value.

For instance - my_name above may be used to initialize another object:

```
INIT another_name
      name = {my_name.name}
END INIT
```

2.5 Repositories

Object Repositories are user-specified directories which contain object descriptions. Object definitions, initialization as well as any information which may be kept in Object Description Files (.d) or DSPACK data files (.ds) may be stored in Object Repositories.

The list of repositories should be stored in the environment variable **DSPACK_REP_PATH** separated by colons and in the order in which they are to be searched.

If a reference to an *undefined* object is made and **DSPACK_REP_PATH** is set then the following procedure is applied:

- a search is made for a file ¡object_name¿.h
- if not found - an error message is produced
- if found - DSPACK checks if file **Repository.ds** is present in the same directory. If found it is read, otherwise file ¡object_name¿.h and, if present ¡object_name¿.d and ¡object_name¿.ds are read.
- the process may be recursive - all undefined objects referred to in the files read will trigger a similar search.

This system is intended to remove any need of defining objects within the programs and also to reduce the need of specifying many names of object description files for each program. Some repositories will probably be common for the collaboration and may be maintained by the CVS system, others may be specific to a program or to the user or to a specific application.

New command *repository* has been added to **ds db** menu. It creates in current directory skeleton repository containing all objects present in memory. all .h files are created as well as Repository.ds file containing all definitions.

2.6 Multiple Servers

Some processes - monitoring, event display and so on - require access to different servers without stopping the program. There is no possibility of one process accessing two different servers at the same time but a new procedure has been introduced to allow the process to disconnect form the current server and connect to another.

```
int ds_reconnect(char *name, int *ierr)
subroutine ds_reconnect(name,ierr)
character*(*) name
integer ierr
```

The program disconnects from the current server and connects to the server *name*. All connections with the current server are broken - all pointers returned by DSPACK calls are no more valid - they may or may not cause segmentation violation as they may point to existing address in the new server. All handles returned by DSPACK calls are not valid anymore, however if they are used together with names they will be reset to the current valid handles.