Gipsy Data Subsystem

J. P. Terlouw – March 1987

1. Introduction

This document presents the software interface based upon the ideas given in *"The new GIPSY data structure for images"* (March 1985). In the time between the writing of that document and this one, the ideas have evolved and changes in the interface specifications were considered necessary, so a complete set of interface specifications is given, which obsoletes all previous specifications. Features of this data structure are: an N-dimensional image store with named axes, one pixel data format: 32-bit floating point, a hierarchical descriptor area with named items. Important design objectives are simplicity of use, generality and speed. A FITS-compatible set of routines is provided.

The interface routines can be divided into a number of classes:

- *Basic* routines for creating files, testing their existence etc.;
- *Coordinate* routines which manipulate coordinate information;
- *Image I/O* routines which read and write image data, very much like the present READXY;
- *Descriptor* interface routines, which can be used to store and retrieve information which accompanies the image data proper.

2. Coordinate representation

2.1. Levels of coordinates

Three levels of coordinates are relevant:

1. *file* coordinates, which are directly related to the position in the file;
2. *grid* coordinates; at this level the meaning of every dimension and the position of the origin are established; axes in this coordinate system are specified symbolically; there is a one-to-one relationship between grid coordinates and file coordinates, though the user need not know anything about this relationship because routines are provided to translate one into the other;
3. *physical* or *astronomical* coordinates; the conversion between these coordinates and grid coordinates is the responsibility of a set of special-purpose coordinate transformation routines.

This level structure forms a strict hierarchy; direct transformations from physical coordinates to file coordinates and the other way around are not allowed.

2.2. Coordinate words.

Coordinate words are nothing more than a packed representation of a vector of file coordinates in the N-dimensional structure. The motivation for this packing is that the resulting coordinate words can be manipulated more easily. The simplest way to do this is summing all coordinates in such a way that the result represents an address in the structure. If we—only for this process of summing—assume a bigger range for the coordinates than is actually necessary for addressing the pixels, then we are able to mark axes with special flags e.g. "this coordinate can vary" or "assume the lowest (or highest) possible value for this coordinate". Having done so enables us to handle defaults in a flexible and general way and we can use a single general format for identifying both pixels and substructures such as lines, planes, boxes etc.

Packing coordinates (into 32 bit) is defined as follows:

$$\begin{aligned} f_0 &= 1, \\ f_i &= f_{i-1}(d_{i-1}+1), \\ C &= \sum_{i=0}^{N-1} c_i f_i. \end{aligned}$$

and unpacking:

$$c_i = \frac{C \bmod f_{i+1}}{f_i}$$

where $d_0 \ldots d_{N-1}$ represent the N dimensions, $c_0 \ldots c_{N-1}$ represent the coordinates, C is the packed coordinate and the relation $0 \leq c_i \leq d_i$ is assumed. To the special case where $c_i = 0$ the following meaning is attached:
If C is used in descriptor access, it means that this coordinate is undefined. If, for instance, in a Westerbork data cube the coordinate velocity is undefined, and the two spatial coordinates are defined, the corresponding coordinate word represents a velocity profile at the specified position. An other (complementary) example is the case where in the same data cube only the velocity is defined, the coordinate word represents a channel map at a specific velocity.
If C is used for addressing data in the N-dimensional structure, it can either mean that the information is not relevant or that extreme values are to be taken, depending on the context in which the coordinates are used.
An implication of this coding is that $C = 0$ designates the whole N-dimensional structure and a range C_1 to C_2 with both C's zero contains all data.

Coordinate words have a number of interesting properties. If they are added, the equivalent of a vector addition is performed; If in such an addition undefined individual coordinates are involved, an intersection of substructures results. Of course these properties should be exploited with care because operations on coordinate words can cause internal overflows in individual coordinates which are not noticed by the computer hardware. For this reason coordinate words should only be manipulated by a set of routines designed for this purpose.

2.3. Ranges and substructures.

Ranges can be specified as two coordinate words; substructures are specified by one coordinate word in which individual coordinates may be undefined. The following conversions are possible:

2 coordinates	→	1 substructure
1 substructure	→	2 coordinates in which the varying coordinates are set to the edges of the N-dimensional structure, i.e. a *range*.
2 substructures	→	1 substructure (intersection)

A set of routines to perform these conversions will be provided. The coordinate word will be used as the main vehicle by which coordinates etc. are transferred between program modules. Coordinates in vector form are not normally needed in application programs because the user communicates coordinate information to a routine which converts the user input to coordinate words.

3. Descriptor structure

The descriptor (or 'header') is a hierarchical structure which allows the user to store information at every possible level by specifying the coordinates of that level. All substructures, from the whole structure down to the pixel level, can have descriptors of its own. This is accomplished by setting the relevant coordinates to an agreed-upon undefined value. If for instance all coordinates of a descriptor item are defined, it must belong to an individual pixel and if all coordinates are undefined, the descriptor item belongs to the whole N-dimensional structure. The descriptor reading routines start looking at the specified level and if the requested information is not found, they inspect higher levels until the information is found or proved to be not present. The descriptor writing routines only write at the specified level.

Descriptor items also have a name; so a descriptor item is fully identified by the combination of name and coordinates. It is allowed that the same descriptor names are used at different levels. This makes it possible to state a general case at a high level and allow exceptions at a lower level.

For application programs, the basic structure of a descriptor item is simply an extendable sequence of bytes, which can be read or written at any position. Access at this level is not recommended for normal application programs. Upon this basic structure a set of interface routines which follows the FITS-standard will be built. These are the routines of choice.

4. Routine descriptions

All routine names have a three- or four-letter prefix: `GDS` for basic routines, `GDSC` for coordinate routines, `GDSD` for descriptor routines and `GDSI` for image I/O routines.

4.1. Basic routines

4.1.1. Create set

`GDS_CREATE` *(file_id, error)*

This routine creates a new structure ("set"). Initially this structure is zero-dimensional. Its shape is determined later by `GDS_EXTEND`.

file_id	File identifier. CHARACTER*(*).
error	Error return code

4.1.2. Test for set existance

`GDS_EXIST` *(file_id, error)*

LOGICAL function; .TRUE. if the set exists; .FALSE. otherwise.

4.1.3. Delete set

`GDS_DELETE` *(file_id, error)*

4.1.4. Extend set

`GDS_EXTEND` *(file_id, name, origin, size, error)*

This routine increases the number of dimensions after having fixed the dimensions already in use.

name The symbolic name of the new axis (CHARACTER*(*)). The FITS convention of naming axes is followed. (≡ CTYPEn)

origin Location of reference pixel along this axis (≡ FITS item CRPIXn). REAL*4.

size The size of the new dimension. Optional argument; if omitted, 1 is assumed. The size of the last created dimension can also be extended automatically when writing data.

4.2. Descriptor routines

This section gives an overview of all descriptor interface routines. Please note that not all are recommended for general use by application programs. Recommended routines are indicated by the symbol ♡.

4.2.1. Write descriptor item

`GDSD_WRITE` *(*
file_id,
name,
level,
buffer,
nbytes,
position,
bytes_done,
error
)

Routine to create a new descriptor item or to write to an existing one. It is possible to write to any part of an existing descriptor item or to append to it; new descriptor items can only be written from the beginning.

file_id File identifier. CHARACTER*(*).

name Name of the descriptor item. CHARACTER*8.

level Coordinate word specifying the level at which the descriptor item is to be placed. INTEGER*4.

buffer Variable or array of any type except CHARACTER containing the data to be written.

nbytes Number of bytes to be written. If the argument is omitted, the default is the current length of the descriptor item. INTEGER*4.

position The relative position, starting at 1, at which will be written. If zero (0) is specified, writing takes place at the end of the descriptor item. If the argument is omitted, the default is 1. INTEGER*4.

bytes_done The number of bytes which have actually been written. INTEGER*4.

error Error return code.

4.2.2. Read descriptor item

```
GDSD_READ   (
            file_id,
            name,
            level,
            buffer,
            nbytes,
            position,
            bytes_done,
            error
            )
```

Routine to read an existing descriptor item. It is possible to read from any part of the descriptor item.

file_id	File identifier. CHARACTER*(*).
name	Name of the descriptor item. CHARACTER*8.
level	Coordinate word specifying the level from which the descriptor item is to be obtained. If it does not exist at the specified level, the routine will inspect higher levels until the item is found or proven to be not present. INTEGER*4.
buffer	Variable or array of any type except CHARACTER receiving the data to be read.
nbytes	Number of bytes to be read. If the argument is omitted, the default is the current length of the descriptor item. INTEGER*4.
position	The relative position, starting at 1, from which will be read. If zero (0) is specified, reading is done from the current reading position. If the argument is omitted, the default is 1. INTEGER*4.
bytes_done	The number of bytes which have actually been read. INTEGER*4.
error	Error return code. In case of success (≥ 0), it will contain the level where the item has been found.

4.2.3. Obtain length of descriptor item

```
GDSD_LENGTH   (
              file_id,
              name,
              level,
              error
              )
```

INTEGER*4 function.

4.2.4. Rewind descriptor item

```
GDSD_REWIND     (
                file_id,
                name,
                level,
                error
                )
```

Set current read position at beginning of item.

4.2.5. Delete descriptor item

```
GDSD_DELETE     (
                file_id,
                name,
                level,
                error
                )
```

Delete descriptor item at the specified level.

4.2.6. Delete at all levels

```
GDSD_DELALL     (
                file_id,
                name,
                error
                )
```

Routine to delete the descriptor *name* at all levels where it exists.

4.2.7. Write FITS item

```
GDSD_WFITS      (                                                    ♡
                file_id,
                name,
                fits_record,
                error
                )
```

Routine to write a FITS record at the specified level.

name name of FITS item. Optional CHARACTER*8 input argument. Default: the keyword found in *fits_record*.

fits_record CHARACTER*80 value, containing a complete FITS header record consisting of keyword, data and comment. The keyword field may be empty (or even illegal) if *name* is specified.

4.2.8. Conditionally write FITS item

```
GDSD_BFITS      (
                file_id,
                name,
                fits_record,
                error
                )
```

Like GDSD_WFITS, except that the descriptor item will only be written when it does not have an *identical* occurence at the specified or at a higher level. This routine is intended for tape-reading programs which append to an existing set.

4.2.9. Read FITS item

```
GDSD_RFITS      (                                                    ♡
                file_id,
                name,
                fits_record,
                error
                )
```

4.2.10. Write FITS item data

```
GDSD_Wtype      (                                                    ♡
                file_id,
                name,
                value,
                error
                )
```

Write to an existing FITS item or create one. In an existing record only the value is changed; the comment field is not affected.

type	REAL, DBLE, INT, LOG or CHAR, depending upon the type of *value*.
value	variable or constant containing the data to be written. REAL*4, REAL*8, INTEGER*4, LOGICAL*4 or CHARACTER*(*).

4.2.11. Read FITS item data

```
GDSD_Rtype      (                                                    ♡
                file_id,
                name,
                value,
                error
                )
```

4.2.12. Write variable length record

```
GDSD_WVAR       (
                file_id,
                name,
                level,
                data,
                error
                )
```

Write a variable length record at the end of a descriptor item. This routine is primarily intended for writing comment or history data.

data CHARACTER*(*) containing the data.

4.2.13. Read variable length record

```
GDSD_RVAR       (
                file_id,
                name,
                level,
                data,
                error
                )
```

Read a variable length record at the current read position.

4.2.14. Find next descriptor name

```
GDSD_FIND       (
                file_id,
                level
                rec_no,
                error
                )
```

CHARACTER*(*) function returning a descriptor name. The purpose of this routine is to obtain all descriptor names at or above a certain level. This is useful when transferring data to a different set or to tape.

level if specified, the routine only finds items of this and of higher levels; if not specified, all items are found.

rec_no Internal record number after which the search starts. Upon return, it will contain the record number where the item was found. If it cannot find an item, zero is returned. INTEGER*4 input/output argument.

error error return code. < 0 indicates an error; ≥ 0 indicates the level where the item was found.

4.3.1. Read image data

```
GDSI_READ   (
            file_id,
            coord1,
            coord2,
            buffer,
            buf_len,
            pixels_done,
            transfer_id
            )
```

General routine to read a substructure in the N-dimensional database. This routine is a generalization of the current subroutine READXY. Note that this routine allows a range of access possiblities by selectively omitting arguments.

file_id Input argument. CHARACTER*(*) identification of the file containing the data.

coord1,2 Input arguments. Two INTEGER*4 coordinate words specifying the part of the structure to be read. This are optional arguments. If omitted, the whole structure will be read; if only the first one is present, a substructure will be read.

buffer Output argument. REAL*4 array to receive the pixels. It should at least accomodate for *buf_len* pixels. Optional argument. If omitted, the routine behaves like a function and delivers one pixel. If in this case *buf_len* $\neq 1$, an error condition exists.

buf_len Input argument. Maximum number of pixels to transfer. Optional argument. If omitted, 1 is assumed. INTEGER*4.

pixels_done Output argument. INTEGER*4 Receives the number of pixels actually transferred. Optional argument. If omitted and *pixels_done* $\neq$ *buf_len*, an error condition exists.

transfer_id Input/output. INTEGER*4. Before the first call, 0 should be assigned to it. When a number < 0 is returned, an error has been detected. A number > 0 means that the transfer has not been completed with this call and that further calls are necessary. These further calls should be made using the unmodified *transfer_id*. If 0 is returned, the transfer has been completed successfully.
Optional argument. If omitted, any condition which would cause a value $\neq 0$ to be returned leads to the activation of an error routine.

4.3.2. Write image data

```
GDSI_WRITE  (
            file_id,
            coord1,
            coord2,
            buffer,
            buf_len,
```

pixels_done,

transfer_id

)

4.3.3. Cancel transfer

`GDSI_CANCEL` *(transfer_id)*

4.4. Coordinate routines

4.4.1. Extract grid coordinate from coordinate word

`GDSC_GRID` *(*

file_id,

axis_name,

coordinate,

error

)

INTEGER*4 function returning the grid coordinate value on a specified axis. If the coordinate value for this axis is not defined, this fact is reflected in *error* and the lower-end value is returned.

axis_name CHARACTER*(*). A unique abbreviation is allowed.

4.4.2. Apply grid coordinate to coordinate word

`GDSC_WORD` *(*

file_id,

axis_name,

grid_coord,

coordinate_in,

error

)

This function constructs a coordinate word from an existing coordinate word and a grid coordinate value. Repetitive calls to this function enable the programmer to construct a completely defined coordinate. It can also be used to perform vector addition and subtraction.

4.4.3. Intersect coordinate words

`GDSC_INTERSECT` *(file_id, coord_1, coord_2, error)*

Function which returns the coordinate word resulting from vector addition applied to two other coordinate words. It is a generalization of `GDSC_WORD`; What `GDSC_WORD` does for one coordinate, is done for all coordinates by `GDSC_INTERSECT`.

4.4.4. Calculate substructure

```
GDSC_SUBSTRUCT  (
                file_id,
                coord_1,
                coord_2,
                error
                )
```

Function to calculate a substructure coordinate word from two coordinate words representing an area. If the area is for instance a two-dimensional piece of a plane, the coordinate word corresponding to that plane is returned.

4.4.5. Calculate range

```
GDSC_RANGE      (
                file_id,
                coord_in,
                coord_out_1,
                coord_out_2,
                error
                )
```

4.4.6. Dimensionality

```
GDSC_NDIMS      (file_id, coordinate)
```

INTEGER*4 function to obtain the dimensionality of a substructure coordinate word.

4.4.7. Axis name

```
GDSC_NAME       (file_id, n, error)
```

CHARACTER function returning the name of the n^{th} axis.

4.4.8. Axis size

```
GDSC_SIZE       (file_id, axis_name, error)
```

INTEGER*4 function.

4.4.9. Axis origin

```
GDSC_ORIGIN     (file_id, axis_name, error)
```

5. User interface

Adopting the new data structure has inevitable consequences for the way in which the user can specify parameters to programs. "Sets" can have names instead of numbers; "subsets" will become part of coordinates; axes will be specified symbolically. The following example shows how the user might specify parameters:

"Behind" the AREA= keyword there will be a new version of the subroutine BOX which eventually delivers one or two coordinate words. It could be made such that it keeps prompting the user until the required dimensionality is reached. The shown input is the simplest possible notation (grid coordinates) but the new BOX could very well be capable of accepting other coordinates. It may be possible to unify BOX with ANYCOO. Such a routine might have the following specification:

```
GETCOO          (
                file_id,
                dimensionality,
                coord_in_1,
                coord_in_2,
                coord_out_1,
                coord_out_2,
                axis_mask,
                default_level,
                keyword,
                message,
                error
                )
```

To make the change to the new system easier, there will be a layer of software which emulates most of the present data structure.

6. Implementation

Image data and descriptor data will be stored in two *separate* files. This decision was necessary because both image and descriptor can be extended. The extra cost compared to one single file consists of the fact that there will be twice as many open files and to a lesser extent the fact that one always should care about two things when moving files using VMS-commands.

6.1. Image data

Image data is stored in a file with extension .IMAGE and contains nothing else but pixel values. To obtain maximum efficiency, the software attempts to allocate reasonably large contiguous pieces of disk space and when doing I/O it tries to transfer as large blocks as possible.

6.2. Descriptor data

The descriptor file has the extension .DESCR. It consists of a fairly large ($\sim$ 1000) number of linked lists consisting of fixed-length records of which the list heads are stored in a table. (A separate linked list, the *freelist*, contains all records currently not in use) This table is addressed by the value of a function which transforms the combination of the descriptor item and the coordinate level in such a way that the results are distributed evenly over all entries of the table. This approach enables very fast retrieval of any descriptor item. To improve the efficiency of attempts to retrieve non-existing descriptor items, the items in any linked list are stored in an ordered (e.g. alphabetic) sequence.

The length of a descriptor item can be arbitrarily large. This feature is implemented by allowing a side-chain of records ("extension records") sprouting from the record containing the item name and coordinate level ("key record"). The key record contains the following information:

- Key record indicator (*true*, changed to *false* when record is moved to the freelist)
- Name
- Coordinate level
- Link to next key record (i.e. the next descriptor item)
- Link to first extension record
- Link to last extension record
- Total number of bytes in this item
- Link to last extension record read
- Sequence number of last byte read
- Descriptor data

The size of a single record should at least accomodate enough space for an ASCII-coded FITS record. If we impose the (not absolutely necessary) restriction that an integral number of records should fit into a 512-byte disk block or virtual memory page, the minimum size would be 128 bytes, of which 99 would be available for data.

Extension records contain only data, a key record indicator (*false!*) and a link to the next record.

When a new record should be allocated, the software first tries to obtain it from the freelist. If this is empty, a record is allocated at the end of the file.

To improve efficiency, the whole structure may be contained in virtual memory during the time that processing takes place. Two possiblities exist: the file can be mapped to virtual memory, possibly in a shared region accessible to multiple processes, or it could be read in entirely after opening and written back before closing. Both methods have advantages and disadvantages which are being evaluated.

The descriptor file consists of two parts: the first part is a header with a size which is a multiple of the record size and the second part is a collection of records. The header part contains the following information:

- Record size
- Number of records allocated (inc. header records)
- List head table size (prime number)
- Free list head
- List head table

CONTENTS