IRAM Memo 2012-?

Preparing GILDAS for large datasets
II - GILDAS Data Format Version 2

S. Guilloteau[1], S. Bardeau[2], J. Pety[2,3]

1. LAB (Bordeaux) 3. IRAM (Grenoble)
3. Observatoire de Paris

29-jan-2014
version 1.1

Abstract

With the advent of large data sets, the original GILDAS Data Format (for hypercubes and tables) became outdated due to its initial pure 32-bit implementation. The advent of new capabilities in interferometers, such as such as on-the-fly mosaics or polarization, also require modification in the way information are stored in UV tables.

A new version of the GILDAS Data Format, called GDFV2, was designed and implemented to answer these challenges. This document details what was done and how it affects both programmers and users (in particular the backward compatibility aspects).

The new data format is detailed. The public FORTRAN application program interface (API) which can be used in programs to to access the data format is described.

Related documents: *GreG documentation*, *Programming in GILDAS*, *Task Programming Manual*

Contents

1 Introduction

The size of the datasets produced by the (sub-)millimeter single-dish and interferometers, including the IRAM 30m, Plateau de Bure interferometers and ALMA, experience a tremendous increase (because of wide bandwidth receivers, spectrometers with thousands of channels, multi-beams, and/or new observing modes like the interferometric on-the-fly).

In the IRAM 30m context, the new FTS backend delivered during 2011 can produce spectra of up to 37275 channels. Combined with On-The-Fly map mode which can contain 10,000's of spectra, the number of individual data values can now easily reach 2 billions. The advent of NOEMA (a major upgrade of PdBI capabilities) will also lead to large data sets.

While CLIC and CLASS data formats have their own limitations (which will be described in another document), imaging through gridding and deconvolution uses the GILDAS data format (hypercubes and tables), which were initially implemented for 32-bit machines. The original implementation of this Gildas Data Format (hereafter GDFV1) is counting the number of (4-bytes) elements it contains in a signed integer value (Integer(4)), which, in IEEE arithmetics, is limited to $2^{31} - 1 \simeq 2.10^9$ elements. Replacing this signed integer value (integer(4)) by a signed long value (integer(8)) would have not helped as the dimensions of the data array are also coded with a signed integer value. The only correct solution was to also store these dimensions as signed long values with effect everywhere in the GILDAS code (both kernel and packages). This way 1D arrays can now contain as much as $2^{63} - 1 \simeq 9.10^{18}$ elements.

In addition to these size limitations, new observing modes, such as polarization or interferometric On-The-Fly, becomes available. Many new requirements appeared to support these modes: Support of more than 4 dimensions, new header parameters, more flexible UV tables, ... Time had come to revise the Gildas Data Format to a new version (hereafter GDFV2).

Section 2 summarizes the GDF design. Sections 3 and 4 describes how the GDF format can store two main kinds of data, namely hypercubes and tables. Changes for end-users and programmers are detailed respectively in Sections 5 and 6. Backward compability and other miscellaneous issues are dealt with in Section 7. For convenience, the data format and the public application program interface of the GIO library is cut and pasted from the code in Appendix A to D. Programmers are encouraged to look directly into the source codes for up-to-date information.

2 Design of the GILDAS Data Format

The Gildas Data Format and its associated API are based on concept of memory copy. In a program, access to a GDF file is done by reading into memory the file header and part or all of the file data. These informations are stored in a Fortran derived type named `gildas`. This derived type is described in Appendix A.

It is essentially build on 5 items:

- the derived type `strings`, which contains character strings from the header
- the derived type `gildas_header`, which contains essentially a direct copy of the file header (except for strings), but in the machine native representation.
- the derived type `loca`, which contains memory related information
- and a few ancillary information about the data, such as the `blc` and `trc` (Bottom Left corner and Top Right corner) which indicate which subset of the data is read, the data filename, etc...

- Fortran pointers as placeholders for the data. These are provided for Real(4) arrays only, and only up to Rank 4. They may or may not be used by the programmer, who can elect to read the data in other Fortran arrays.

While in memory, quantities are always handled in the native machine format, in the data file, numbers may be represented in a different way. The API (avaible is the so-called GIO library) handles the data format conversion (in the data and in the header) transparently. The data format, although described here, is **not** intended to be used directly, but only through the API.

3 GDF Images

The Gildas Data Format can handle different type of data. In GDFV1, these were Images, Tables and UV Tables, but the distinction between Images and Tables was limited, as any Image could be opened as a Table. In GDFV2, we have improved this distinction by allowing different kinds of Tables (simple tables, UV Tables, Class Tables, VO-like Tables), which are identified through a specific code in the data header.

3.1 Type description

A GDF file is a binary file starting with a *header* spreading over *header blocks*. The *data* is found after the *header*, divided in its own *data blocks*. We briefly describe here the GDFV2 structure; the changes between GDFV1 and GDFV2 and the backward support are detailed later on.

3.1.1 Header description

The header is divided in sections, as follows. The exact header definition can be found in appendix B.

1. Leading information: vital information describing the file format.
2. DIMENSION: describes the dimensions of the data in the file.
3. BLANKING: the blanking and tolerance values.
4. EXTREMA: the value and position of the minimum and maximum value in the data.
5. COORDINATE: the axes definition for each dimension, encoded as a 2D array of N_{dim} (ref,val,inc) triplets.
6. DESCRIPTION: the data unit and axes names. The first header block ends here.
7. POSITION: the source description. The second header block starts here.
8. PROJECTION: the projection definition.
9. SPECTROSCOPY: the description of the spectroscopic axis.
10. RESOLUTION: the beam characteristics.
11. NOISE: the data noise.
12. ASTROMETRY: proper motion parameters.

Table 1: GDF encoding of the system and version. Most of the recent computers are little endian (IEEE) machines. The obsolete system VAX is not supported with GDFV2. 1: hyphen, 2: underscore.

Version	IEEE	EEEI	VAX
1	-[1]	.	_[2]
2	<	>	N/A

Table 2: List of kinds available for GDFV2 files (`h%gil%type_gdf`). The values are integers, but should not be used explicitely.

Name	Value	Comment
code_gdf_image	0	Normal data, i.e. IMAGE...
code_gdf_uvold	1	Old UV Data with wrong weight
code_gdf_uvt	10	UV Data in "visibility" order
code_gdf_tuv	-code_gdf_uvt	The transposed way (channel order)...
code_gdf_table	20	A simple Table, with no information
code_gdf_vo	-code_gdf_table	Virtual Observatory (VO) Tables are ordered in the transposed way...
code_gdf_xyt	40	A specific CLASS table (by symmetry with code_gdf_uvt)
code_gdf_txy	-code_gdf_xyt	A specific CLASS table (by symmetry with code_gdf_uvt)
code_gdf_sheet	50	A spread sheet for the SIC Tabler...

13. UV_DATA: support for UV tables, e.g. description of extra-columns specific to this kind of tables.

The leading pseudo-section and thus all the GDF files (V1 or V2) start with 12 specific characters:

- the 6 firsts are always the word `GILDAS` (uppercase),
- the 7^{th} is a unique character encoding the system type and the version of the GDF used (see Table 1),
- the 5 last are a subsequent word used to recognize the kind of data in the file, namely `IMAGE` (for standard Gildas images or N-cubes) or `UVFIL` (for Gildas UV tables).

In GDFV2, this last character string is only a first order information about the file kind. Distinction between Images, Tables, UV Tables, VO-like Tables, is provided by an integer keywords (see Table 2).

Then follows the data format, the various number of blocks in the file, the version of the GDF currently in use, and the kind of GDF file.

Each section has also a parameter describing its length. A zero value means that this section is not present in the file and/or not filled in memory. For the programmer or the end-user, any other value means it is enabled (then all the parameters in the section are expected to be filled).

In details, some of the sections have a fixed length for any file (e.g. BLANKING is 2 + 2 4-bytes words) and some have a length depending on the number of dimensions in the file (e.g.

COORDINATE is $2 + 2 \times (N_{dim} \times 3)$ words).

3.1.2 Data description

The kind of data stored is described by the parameter `form` in the leading information of the header (see section 3.1.1). In most of the radioastronomy applications, single precision floats are sufficient: the code `fmt_r4` is used by default, but double precision floats (`fmt_r8`), standard (`fmt_i4`) and long integers (`fmt_i8`), and single precision complex (`fmt_c4`) are also supported.

As for the header parameters, the values are stored in the native system of the machine (little endian or big endian) at the file creation time. The codes detailed in table 1 ensure that the values can be re-read and updated later on any system.

The data is stored in the file in column-major order (Fortran-like). Its dimensions are described by the *Dimensions* section. While using standard (32 bits machines) or long (64 bits) integers when in memory, they are always stored in the file as long integers: the limit is then $2^{63} - 1 \simeq 9.10^{18}$ elements per dimension. The total number of elements in the data uses the same limit. Note that cubes larger than 2 GB (512 mega-elements of single precision floats) reach the RAM limit of 32 bits machines. This means that the whole data block cannot fit in memory. Such big data files (> 2 GB) are **not** supported on 32 bit machines, as these machines are now obsolescent.

The parameter `nhb` indicates the number of header blocks, and `ndb` that of data blocks. Parameter `ntb` has been provisioned to handle trailing blocks after the data. All blocks are 512 Bytes long in the current version. The total number of blocks `nhb+ndb+ntb` is rounded (upwards) to a multiple of 16, so that reading can proceed with physical blocks of 8192 Bytes for better efficiency. A particular effort has also been made here to ensure that there is no Integer(4) limit on the number of Fortran records (blocks) in the file: all record computations and numbers use Integer(8) values.

3.2 Improvements between GDFV1 and GDFV2

3.2.1 Dimensions

The maximum number of dimensions supported *by the GIO library*[1] is now coded as a unique parameter named `GDF_MAXDIMS`; it has been increased from 4 to 7[2] between the two GDF versions. The SIC arithmetic engine has been prepared during the summer 2011 to be able to deal with such data cubes. Increasing this value is technically possible at the cost of hand-made adjustments of the GIO library, of the SIC arithmetic engine, maybe of the tasks and procedures, and probably on the global efficiency.

The maximum number of dimensions supported *by the GDFV2* is also set to `GDF_MAXDIMS`. Increasing it to 8 is technically possible but useless without the GIO and SIC libraries also upgraded. 9 dimensions and beyond can not be currently supported by the GDFV2, since the `DESCRIPTION` section would overflow the first header block.

In every case (GDFV2, GIO, SIC etc), increasing the maximum number dimensions beyond 7 implies the support by the compilers of the rank 15 arrays from the Fortran 2008 standard. This, because we tend more and more to map the GDF or SIC data block on Fortran arrays.

[1]including the GDF↔FITS converter

[2]Up to the 2003 standard, Fortran used the same limit on its arrays. Fortran 2008 has raised it to 15.

In the GDFV1, the number of elements per dimension and the total number of elements in the cube were stored in a standard integer, limited to $2^{31}-1$. In the GDFV2, these elements are now stored as long integers (under 64 bits machines) and are now limited to $2^{63}-1$.

3.2.2 Transposition of data N-cubes

The transposition engine has been clearly isolated from the GIO (for cube transpositions) and SIC (for variable transpositions) libraries, and has been moved to the GMATH one. It benefits several improvements:

- it supports also 7 dimensions (but could support more, independently of GIO or SIC),
- timings and duration projections have been added for user feedback when transposing larges cubes,
- more transposition codes are supported, including all the 3D permutations.

3.2.3 New header parameters or facilities

A number of new objects have been added to the header in addition to all the previous ones:

- For the programmer convenience, 3 pointers to the `ref(:)`, `val(:)` and `inc(:)` values are now available, allowing easy writing of loops on the dimensions. They are real Fortran pointers to (and not duplicate of) the `convert` array from the COORDINATE section. [3]
- In the EXTREMA section, the min and max value coordinates in the cube are coded in memory as a `MINLOC` and `MAXLOC` arrays with 7 elements. In the disk file, they are coded as 2 *flat* (scalar) address in the data.
- The Doppler factor (`dopp`) and the velocity type (`vtyp`) have been added to the SPECTROSCOPY section.

3.3 Backward compatibility

The GDFV2 is an improved and enlarged version of the GDFV1, but the GDFV1 is not directly compatible with the GDFV2. Changes for programmers and end-users are described in the sections 5 and 6 respectively.

Reading the GDFV2 file structure from disk with an old version (older than May-2012) of Gildas is not possible: the file will not be recognized and the following error message will be returned:

```
E-RIS,  File xxx.gdf is neither a GILDAS data frame nor a SIMPLE FITS file
```

Reading the GDFV1 file structure from disk (in any format, including VAX) is still possible; it is mapped on the fly to a GDFV2 structure in memory. On the other hand, writing GDFV1 is not possible anymore.

For convenience, during a transition phase, some update operations remain possible. Extending a GDFV1 data file, for example (via the `DEFINE IMAGE A[n] gdfv1.gdf EXTEND` command),

[3]Variables `%gil%refI` with `I=1,4` (ibid. `%gil%valI %gil%incI`) which were marked as deprecated, have been removed. The new pointer arrays should be used instead.

is still possible. Updating the extrema is also possible. Similarly, extending a pre-existing GDFV1 UV Table in CLIC is allowed.

These specific facilities are only provided to ease the transition between GDFV1 and GDFV2, and will ultimately be removed from the GILDAS package. The user is thus encouraged to convert its data files to the new format. The simple SIC procedure `@ gag_pro:gdf_convert` will do this.

4 GDF UV Tables

4.1 Why a new implementation of the UV Tables in GILDAS ?

The UV Tables in the original GILDAS V1 data format, created in 1989, handled visibilities and their required associated data parameters in a very similar way as the UVFITS data format used to export data from the VLA at that epoch. A *visibility* consisted in
- 7 associated data parameters (*daps*), in fixed order: `u, v, w, date, time, first antenna, second antenna`
- for each channel, a real part, and imaginary part, and a weight (a so-called *complex visibility element*).
The *daps* precede the *complex visibilities elements*.

The 7 (*daps*) were real numbers. As Plateau de Bure is a 2-D array with small field of view, `w` was occasionally replaced by the scan number `scan` for debugging purpose. The number of channels was simply derived from the visibility size:

$$Nchan = (VisiSize - 7)/3 \quad (1)$$

Early 2000, an extension was made to this constraining layout by optionally adding two (real) numbers after the channel values. This was made necessary by the development of the ALMA simulator to study the impact of the compact array (ACA).

With further developments coming on, such as polarization, on-the-fly mosaicing, multi-source UV FITS files from CASA, etc... this scheme was becoming far too constraining. In addition, UV data sets could readily exceed the 2 GByte size limit of the GILDAS V1 data format. We decided to take advantage of the changes required to handle large data sets to implement a more flexible UV data format.

4.2 The new layout

4.2.1 Data Format Identification

UV Tables are now a very special subset of GILDAS data files. As any GILDAS data file, they are identified by an integer code stored in variable `h%gil%type_gdf` (see Table 2) which, for UV Tables must take the value `code_gdf_uvt` or `code_gdf_tuv`, depending whether the table is in native order (a visibility being contiguous in memory) or transposed. For simplicity, we have imposed `code_gdf_tuv = -code_gdf_uvt`.

The new layout differ in several ways from the older one. In short:

- the `daps` are no longer at pre-defined positions in the data file, but their positions are described by the UV table header.
- the *complex visibility elements* are not necessarily (real, imaginary, weight) triplets, but contain `h%gil%natom` elements. For example, a zero spacing UV table from a single-dish spectrum may have `h%gil%natom = 2` (as the imaginary part is 0 in such a case).
- Provision is made to handle polarization data, by indicating the number of Stokes parameters in a visibility, `h%gil%nstokes`.
- The number of *frequency* channels `h%gil%nchan` and the number of visibilities `h%gil%nvisi` are available independently of the transposition status of the UV data set.

The UV specific information is included in the `uvda` section of the `h%gil` header, as described in appendix B.

4.2.2 Dap positions and Size

Each `dap` is identified by a specific code from the list available in appendix C. The `dap` position as well as its length can vary. For example, the `item` dap would be located at `h%gil%column_pointer(code_uvt_item)`, and it could be a Real(4) or Real(8) number depending on the value of `h%gil%column_size(code_uvt_item)`.

A whole set of possible `daps` entities is defined below, and can be extended if needed. The `daps` element can be located either before the complex visibilities, or after, but not intermixed with them. Accordingly, there can be `h%gil%nlead` leading columns before the complex visibility values and `h%gil%ntrail` trailing columns after. Note that these number indicates the number of 32-bit columns, so if any of the `dap` is a 64-bit number (`h%gil%column_size(code_uvt_dap) = 2`), `nlead+ntrail` will be greater than the number of non-zero values in `h%gil%column_size(:)`.

The existence of `daps` entities of different lengths raise a specific issue when transposing a UV tables. By default in GILDAS, transpositions are applied on the global data type. For UV tables, this is `fmt_r4`, i.e. 32 bit entities. Assume we start with a natural (UVT) order, with for example u a 64 bit quantity. After transposition to TUV order, the first 32 bit of u are in the first column, and the last 32 bits in the second. This is incoherent. Thus a second transposition is required on these specific columns to establish the contiguity of the 64-bit values. This is handled transparently by the GIO library, in `gdf_transpose` as well as in `gdf_read_gildas`.

Restriction on Real(8) daps The need for this capability was driven to provide simpler handling of multi-source datasets coming from CASA (which have the RA and Dec of the sources given for each visibility).

However, it is **NOT** intended to be of general use. In practice, it will be reserved for RA and Dec daps, when needed, and UV Tables having such daps are intended to be transformed (split into independent sources, or converted to a multi-field mosaic with only offsets stored) by dedicated applications. For example, multi-source datasets coming from CASA would be imported as FITS files, and converted to GILDAS UV Tables by the GIO library. The resulting GILDAS data sets can then be exploded to single-source UV data very simply.

The capability of having Real(8) also offers more flexibility for the future. However, most daps will still be Real(4) only for the time being.

Are Real(4) U,V,W coordinates sufficient ? A peculiar case are the U,V,W coordinates. With very long baselines (20 km with ALMA) and short wavelengths (0.3 mm), one is tempted to say that U,V,W should be Real(8), as absolute baseline lengths are measured (and defined by the ultimate antenna stability) to a fraction of wavelength, say 40 μm for example. 40 μm/ 20 km equals $2\,10^{-9}$ which exceeds the precision of Real(4) numbers by 2 decimals...

So in principle, Real(4) are insufficient. But this is only true if a full astrometric calibration is required. GILDAS UV tables are intended for imaging purposes only, although this should include self-calibration.

Real(4) precision will only limit the field of view for a given angular resolution. The phase error due to a (relative) numerical precision δ is given by

$$\Delta\phi = 2\pi u \delta \Delta X = 2\pi \frac{B}{\lambda} \delta \Delta X \tag{2}$$

so to obtain for example less than 0.1 radian (6 degree), we must have

$$\Delta X \leq \frac{0.1\lambda}{2\pi B \delta} \tag{3}$$

Plugging extreme numbers ($\delta = 10^{-7}$, $B = 20$ km, $\lambda = 0.3$ mm) we obtain $\Delta X \leq 2.4\,10^{-3}$ radian, or 0.13°, which is quite a wide field of view.

So, the Real(4) precision do not allow to perform normal calibration (phase calibrators being in general more than a (few) degree(s) away), but is quite sufficient for even wide field imaging and self-calibration.

4.2.3 Complex Visibilities

The complex visibility values are always Real(4). The range of 32-bit columns for the complex visibility values is indicated by `h%gil%fcol` and `h%gil%lcol`. We thus have the following relations
`h%gil%fcol` = `h%gil%nlead+1`
`h%gil%lcol` = `h%gil%dim(visi_axis) - h%gil%ntrail`
`h%gil%nlead+h%gil%ntrail` = `sum (h%gil%column_size)`
where `visi_axis` is 1 for the "natural" (UVT) order, in which each visibility is contiguous, and 2 for the "transposed" order (TUV).

The UV data format is in fact a generic telescope table format, and can also handle tables from a Single dish telescope. As weights are often common to all values, the complex visibility can be made of only 2 "atoms" provided `h%gil%column_pointer(code_uvt_weig)` $\neq$ `0`. It can also have only 1 atom, if the imaginary part is always 0. The "atom" types are explicitely described by the h%gil%atoms(1:4) array. Currently defined atom types are

```
!
! Visibility atom description
integer(kind=4), parameter :: code_atom_real = 1
integer(kind=4), parameter :: code_atom_imag = 2
integer(kind=4), parameter :: code_atom_weig = 3
```

Other codes may be added if needed, for example integration time (as Tsys weighting is not necessarily the thing to do when handling different polarizations).

4.2.4 Polarization Data

This can be handled in three different ways. In the first mode, a single visibility can have only one polarization state, but different visibilities may have different states. The polarization state for each visibility is defined by `h%gil%column_pointer(code_uvt_stok)`. Possible values are given by the `code_stokes_...` parameters specified below

```
! Polarimetry parameters
integer(kind=4), parameter :: code_stokes_i = 0
integer(kind=4), parameter :: code_stokes_q = 1
integer(kind=4), parameter :: code_stokes_u = 2
integer(kind=4), parameter :: code_stokes_v = 3
integer(kind=4), parameter :: code_stokes_ll = 11
integer(kind=4), parameter :: code_stokes_rr = 22
integer(kind=4), parameter :: code_stokes_lr = 12
integer(kind=4), parameter :: code_stokes_rl = 21
integer(kind=4), parameter :: code_stokes_hh = -11
integer(kind=4), parameter :: code_stokes_vv = -22
integer(kind=4), parameter :: code_stokes_hv = -12
integer(kind=4), parameter :: code_stokes_vh = -21
```

while antenna polarizations are specified by the `code_polar_...` parameters.

```
integer(kind=4), parameter :: code_polar_h = -1
integer(kind=4), parameter :: code_polar_v = -2
integer(kind=4), parameter :: code_polar_l = 1
integer(kind=4), parameter :: code_polar_r = 2
```

This mode is indicated by `h%gil%nstokes = 1` and `h%gil%order = 0`. It will also have `h%gil%nfreq = 0` (see below). Note that there is no provision for mixed (circular / linear) polarization states.

In the second mode, a single visibility can handle several stokes parameters, but Stokes and Channels are regularly ordered. In this mode, channels are regularly spaced in Frequency, like in single Stokes mode, so no `h%gil%freqs` array is specified. Stokes parameters are also ordered: for every channel, the ordering of Stokes parameters must be the same. Stokes parameters may vary first, and Frequencies next (the so-called `h%gil%order = code_stok_chan` ordering) or Frequencies may vary first, and Stokes next (the so-called `h%gil%order = code_chan_stok` ordering). Ordering is stored in variable `h%gil%order`. Arrays `h%gil%stokes(h%gil%nstokes)` indicate the Stokes parameter for each frequency "channel". This mode is indicated by `h%gil%nstokes` $>$ `1` and `h%gil%order` $\neq$ `0`, but still has `h%gil%nfreq = 0`.

In the third, most complex mode, the Stokes parameter can vary arbitrarily within one visibility. Then each *complex visibility element* must have a Stokes parameter and a frequency indication, which are given in the arrays `h%gil%stokes` and `h%gil%freqs` respectively, of size `h%gil%nfreq = h%gil%nchan * h%gil%nstokes`. As each *complex visibility element* is given a specific frequency, this mode can also allow to store irregularly spaced frequency channels. Such a mode can be convenient for continuum bandwidth synthesis, for example. Proper weighting is ensured even though the channel width is not specified, as the weights are carried along with the Real and Imaginary parts. This mode is indicated by `h%gil%nfreq` $>$ `0`. `h%gil%order` is irrelevant in this mode, but should be set to 0.

5 Changes for end-users

The changes for users are minimal. As usual, compatibility has been a major design goal in GILDAS. The new software can read transparently old GDF files, including old UV tables. For UV data, it assumes old UV tables contain `SCAN` as the third *dap*, as this was the classical behaviour for tables produced by CLIC.

5.1 Behaviour

The SIC command `V\HEADER /EXTREMA` and the task `EXTREMA` have now a different behaviour for UV Tables than for other GILDAS data sets: they compute the minimum and maximum baseline lengths, which is a more useful quantity in this case than the overall extrema of the data array.

5.2 SIC variables

Some SIC variables mapping the file header (of any kind) have changed:

```
Head%DIM[:]      is now a long integer array of dimension 7,
Head%WHERE[1]    and
Head%WHERE[2]    "flat" positions have been removed, replaced respectively by
Head%MINLOC[:]   and
Head%MAXLOC[:]   array coordinates in the datacube.
Head%GENE        is now the length of the new Dimension section *only*.
```

The new UV data format being more powerful, this is also reflected in SIC. The command `DEFINE UV` checks more strictly the type of data. It defines the following additional SIC variables

```
Head%BASEMIN    Minimum baseline
Head%BASEMAX    Maximum baseline
Head%NVISI      Number of visibilities
Head%NCHAN      Number of channels
Head%NSTOKES    Number of Stokes parameter
Head%NATOM      Complex visibility size
```

In addition, for transposed UV tables, `Head%U` and `Head%V` arrays are defined to point towards the U and V columns.

5.3 New facilities

Some tasks or commands may access indifferently UVT or TUV tables; implicit transposition is then done if needed. This is currently the case for the command `READ` of Mapping. See section 3.2.2 for details on the new capabilities of the transposition engine.

6 Changes for programmers

This is obviously the biggest part. For images, the new format does not provide additional features, so the changes are small, and limited to the fact that dimensions and sizes are now Integer(8) integers. For Tables however, the new format is much more flexible, and thus more difficult to handle without problems. In the context of UV tables, three features can cause unanticipated problems if they are ignored or mis-used: the possible existence of Real(8) `dap` elements, the variable number and ordering of `daps`, and the multiple Stokes values or irregularly spaced frequency channels.

We have tried to simplify the programming in several ways, by providing a more convenient data structure and a comprehensive API.

6.1 Dealing with GDF files (any kind)

No access the Gildas derived type elements should be done before calling the subroutine `GILDAS_NULL`, as some elements are dangling pointers.

```
call gildas_null(hx, type)
```

where `hx` is the Gildas header to be initialized, and `type` a character string indicating the desired type of Gildas data structure. `type` is an optional character string. Recognized values are `'IMAGE'`, `'TABLE'`, `'UVT'`, `'TUV'` and `'VOTABLE'`, in link to the types described in Table 2. The default is set to `'IMAGE'`.

- The 4 triplets `(refj,valj,incj)` (with $j = 1$ *to* 4) have been removed. For a more generic approach with 7 dimensions, these 4×3 scalars have been replaced by 3 vectors `ref(:), val(:), inc(:)` (as described in section 3.2.3).
- Similarly, the 4 pairs `(minj,maxj)` (with $j = 1$ *to* 4) have been removed. These 4×2 scalars have been replaced by 2 vectors `minloc(:)` and `maxloc(:)`.
- each section length variable has been renamed as detailed in Table 3 (usually by adding the suffix `_words` to the GDFV1 name). The *General* information has been split into the 2 sections *Dimension* and *Coordinate*. As described in section 3.1.1, any non-zero value is now valid to indicate that a section is enabled for reading or writing.

6.2 UV data format usage restrictions

The Gildas UV tables signatures (12 first characters of the binary file) `GILDAS_UVDAT` and `GILDAS_UVSOR` are not used anymore[4]. The initialization by `GILDAS_NULL` sets the generic signature `GILDAS_UVFIL` instead.

To simplify our life, we have imposed the following constraints

- There must be 7 leading `daps` before the visibilities. These `daps` have to be in sequence: `code_uvt_u, _v, _w` *or* `_scan, _date, _time _anti, _antj`. This *default* ordering of "mandatory" `daps` follows the GILDAS V1 convention.
- The 7 leading `daps` must be real.
- Usage of Real(8) `daps` is restricted to `code_uvt_ra, code_uvt_dec`.

[4]this restriction is actually older than the GDFV2 definition.

Table 3: Renaming of section length variables between the GDFV1 and GDFV2. Absent name indicate a section that has appeared or disappeared.

Section	GDFV1	GDFV2
General	`gene`	-
Dimension	-	`dim_words`
Blanking	`blan`	`blan_words`
Extrema	`extr`	`extr_words`
Coordinate	-	`coor_words`
Description	`desc`	`desc_words`
Position	`posi`	`posi_words`
Projection	`proj`	`proj_words`
Spectroscopy	`spec`	`spec_words`
Resolution	`reso`	`reso_words`
Noise	`sigm`	`nois_words`
Astrometry	`prop`	`astr_words`
UV Data	-	`uvda_words`

6.3 GDF API

Beside the definition of the GDFV2, the GDF API has been revised and simplified. It is listed in appendix D. As the API provides many tools, some redundancy exists, and one may legitimately ask which is the best way to do things. As usual, the answer depends on what you are doing, and also pretty much on the actual size of your UV data sets. Here are some guidelines.

- Developing a new task.
  Use the integrated API `gdf_read_gildas` as much as possible. It provides consistency checks which you do not have to duplicate. The drawback is that it reads all data at once.
- Consolidating a task.
  If your task requires some consistency checking before doing computations, consider using the integrated API `gdf_read_gildas` with optional argument `data = .false.`, and read the data later. The drawback is that you have to allocate the data array yourself. This can be done through `gdf_allocate`.
- Accessing indifferently a .UVT or .TUV data set.
  Use the integrated API `gdf_read_uvdataset` as much as possible. To do so, you must use `gdf_copy_header` or `gdf_transpose_header` to generate the output header from the input one, depending on whether a transposition is required or not.
- Handling large data sets.
  If your application is likely to need to handle large data sets, you should make use of the abilities of the GDF API to read GILDAS files by blocks through the `blc(:)` and `trc(:)` arrays. Many operations are actually sequential, so this can use much less memory, and can be faster than considering the whole data set.

6.4 Obsolete routines

The following routines have been removed.

```
gdf_blis    Change Starting Block. Not used anywhere...
gdf_sris    Set Read status. Not used anywhere
gdf_read    Obsolete step from the original gdf_readx, y or z
gdf_writ    Obsolete step from the original gdf_writx, y or z
gdf_ch      Obsolete step from the original gdf_chxy and others...
```

These ones have become purely internal and been renamed to `gio_XYis` when applicable.

```
gdf_cris    CReate Image Slot: now purely internal gio_cris
            Only used by SIC/defvar.f90
gdf_geis    GEt Image Slot:
            Only used by SIC/defvar.f90
gdf_gems    GEt Memory Slot: only the gio_gems equivalent
            Only used by SIC/defvar.f90
gdf_reis    REad Image Slot: now purely internal
            Only used by SIC/defvar.f90
gdf_clis    CLose Image Slot: now purely internal
            Only used by SIC/defvar.f90
gdf_wris    WRose Image Slot: now purely internal
            Only used by SIC/defvar.f90
gdf_exis    EXtend Image Slot: now purely internal, use gdf_extend_image
            Only used by SIC/defvar.f90
gdf_crws    CReate Work Slot: now available by default on any "image".
            Only used by SIC/defvar.f90
gdf_fris    Use gdf_close_image instead...
            Only used by SIC/defvar.f90 & delvar.f90
gdf_frms    Free Memory Slot, purely internal now.
            One case in SIC/defvar.f90 to be checked...
gdf_lsis    List Status of Image Slot
            Only used by SIC for debugging.
gdf_rhsec   Read Header SECtion: now purely internal
gdf_whsec   Write Header SECtion: now purely internal
gdf_get_data  Use gdf_read_data instead, with an array allocated (e.g.
              by gdf_allocate) and set the loca%addr to point towards
              the allocated array.
gdf_read_image
gdf_free_image
gdf_create_mapped_image
gdf_upih
```

There are a few obsolescent routines, awaiting renaming or deletion.

```
gdf_stis      Test Read/Write status. Only used by SIC.
gdf_geih      GEt Image Header. Only used by SIC.
gdf_flih      FLush Image Header. Only used by SIC.
```

7 Miscellaneous and pending items

7.1 Compatibility

In the development phase, the new GIO library is still able to write down the old data format. Besides the inherent limits of the GDFV1 format (4 dimensions only, $<$ 2 GBytes only), this means a number of additional restrictions.

For UV tables, in particular, there must be no non-standard column (i.e. the only authorized non-zero `h%gil%column_pointer` are those in the range 1-7), no polarization information or irregularly spaced channels.

This facility is provided for debugging only, but not intended to be made available to general users. It is controlled by the logical name `GILDAS_HEADERS` (which is used by all tasks) and, in SIC driven programs, also by command `SIC HEADERS`, so that one can temporarily write in the old format if needed. This capability will become obsolete once the transition phase to GDFV2 is complete.

7.2 Pending issues

A number of items in the GDFV2 header have been kept for backwards compatibility only. They may change in the future, when writing the GDFV1 format will no longer be supported.

- `%loca%getvm`
  This is clearly obsolescent, and perhaps can be removed from the gildas derived type. Only SIC is likely to use this now. No application do so any longer.

- `%loca%addr`
  This is now only used internally in SIC and by a few routines in CLIC, and one task of CLASS (`map_ekh`. They are defined by applying `locwrd` to the appropriate array.

  The idea is to suppress this at some point.

- **Rank of data set**
  A problem with the current code is the variable rank of the data. It would be convenient to use the new Fortran 2003 to have always a rank-1 array declared, and using the Fortran 2003 capability of reshaping through a different rank pointer using this rank-1 array as a target. e.g. for a 2-D Real image:

  ```
  h%r2d(1:h%gil%dim(1), h%gil%dim(2)) => h%real
  ```

  Allocation would be only of the `h%real` array, so that automatic deallocation could be easily made when freeing the image. At least, we would *always* know the data array name when no interface is required...

  Unfortunately, this is only possible with the latest version of the compilers being used so far (ifort 12.0 and gfortran 4.8.0).

  In the meantime, tools have been developped to adjust the rank of an image to the user need when possible (see `rank=` optional argument in `GDF_READ_GILDAS` and `GDF_READ_HEADER`.

A GILDAS Fortran derived type

```
type :: gildas
  sequence
  character(len=filename_length)   :: file = ' '  ! File name
  type (strings)       :: char                    !
  type (location)      :: loca                    !
  type (gildas_header_v2) :: gil                  !
  integer(kind=index_length) :: blc(gdf_maxdims) = 0       ! Bottom left corner
  integer(kind=index_length) :: trc(gdf_maxdims) = 0       ! Top right corner
  integer(kind=4) :: header = 0                   ! Defined / Undefined
  integer(kind=4) :: status = 0                   ! Last error code
  real,         pointer :: r1d(:)       => null() ! Pointer to 1D data
  real(kind=8), pointer :: d1d(:)       => null()
  integer,      pointer :: i1d(:)       => null()
  real,         pointer :: r2d(:,:)     => null() ! Pointer to 2D data
  real(kind=8), pointer :: d2d(:,:)     => null()
  integer,      pointer :: i2d(:,:)     => null()
  real,         pointer :: r3d(:,:,:)   => null() ! Pointer to 3D data
  real(kind=8), pointer :: d3d(:,:,:)   => null()
  integer,      pointer :: i3d(:,:,:)   => null()
  real,         pointer :: r4d(:,:,:,:) => null() ! Pointer to 4D data
  real(kind=8), pointer :: d4d(:,:,:,:) => null()
  integer,      pointer :: i4d(:,:,:,:) => null()
end type gildas

type :: strings
  sequence
  character(len=12) :: type    = 'GILDAS_IMAGE'  !     Image Type (see ITYP)
  character(len=12) :: unit    = ' '             !     Data Units (see IUNI)
  character(len=12) :: code(gdf_maxdims) = ' '   !     Axis codes (see ICOD)
  character(len=12) :: syst    = ' '             !     Coordinate system (see ISYS)
  character(len=12) :: name    = ' '             !     Source name (see ISOU)
  character(len=12) :: line    = ' '             !     Line name (see ILIN)
  ! For Even GDF_MAXDIMS only ! character(len=4)  :: pad_char
end type strings

type :: location
  sequence
  integer(kind=address_length) :: addr = 0  ! Address of image
  integer(kind=size_length) :: size = 0     ! Size of image
  integer :: islo        = 0                ! Image Slot number
  integer :: mslo        = 0                ! Memory Slot number
  logical :: read        = .true.           ! ReadOnly status
  logical :: getvm       = .false.          ! Memory / File indicator
  integer(kind=size_length) :: al64 = 0     ! Padding required for alignment
end type location
```

B GDFV2 header

```
type :: gildas_header_v2
  sequence
    ! Trailer:
    integer(kind=4) :: ijtyp(3) = 0       !  1  Image Type
    integer(kind=4) :: form    = fmt_r4   !  4  Data format (FMT_xx)
    integer(kind=8) :: ndb     = 0        !  5  Number of blocks
    integer(kind=4) :: nhb     = 2        !  7  Number of header blocks
    integer(kind=4) :: ntb     = 0        !  8  Number of trailing blocks
    integer(kind=4) :: version_gdf = code_version_gdf_current !  9  Data format Version number
    integer(kind=4) :: type_gdf =  code_gdf_image  !  10 code_gdf_image or code_null
    integer(kind=4) :: dim_start = gdf_startdim    !  11  Start offset for DIMENSION, should be
    integer(kind=4) :: pad_trail
    ! The maximum value would be 17 to hold up to 8 dimensions.
    !
    ! DIMENSION Section. Caution about alignment...
    integer(kind=4) :: dim_words   = 2*gdf_maxdims+2  ! at s_dim=17  Dimension section length
    integer(kind=4) :: blan_start !! = dim_start + dim_lenth + 2   ! 18  Pointer to next sectio
    integer(kind=4) :: mdim    = 4        !or > ! 19  Maximum number of dimensions in this data
    integer(kind=4) :: ndim    = 0        ! 20  Number of dimensions
    integer(kind=index_length) :: dim(gdf_maxdims) = 0     ! 21  Dimensions
    !
    ! BLANKING
    integer(kind=4) :: blan_words = 2     ! Blanking section length
    integer(kind=4) :: extr_start         ! Pointer to next section
    real(kind=4) :: bval = +1.23456e+38   ! Blanking value
    real(kind=4) :: eval = -1.0           ! Tolerance
    !
    ! EXTREMA
    integer(kind=4) :: extr_words = 6    ! Extrema section length
    integer(kind=4) :: coor_start !! = extr_start + extr_words +2     !
    real(kind=4) :: rmin    = 0.0         ! Minimum
    real(kind=4) :: rmax    = 0.0         ! Maximum
    integer(kind=index_length) :: minloc(gdf_maxdims) = 0 ! Pixel of minimum
    integer(kind=index_length) :: maxloc(gdf_maxdims) = 0 ! Pixel of maximum
    !
    ! In data file, minloc and maxloc are not written. Instead, two Integer(8)
    ! mini and maxi indicate the position of the extrema in a rank-1 model of the data.
    !
    ! COORDINATE Section
    integer(kind=4) :: coor_words  = 6*gdf_maxdims   ! at s_coor  Section length
    integer(kind=4) :: desc_start  !! = coord_start + coord_words +2     !
    real(kind=8)    :: convert(3,gdf_maxdims)   !  Ref, Val, Inc for each dimension
    !
    ! DESCRIPTION Section
    integer(kind=4) :: desc_words  = 3*(gdf_maxdims+1) ! at s_desc, Description section length
    integer(kind=4) :: null_start  !! = desc_start + desc_words +2     !
```

```
integer(kind=4) :: ijuni(3)   = 0        ! Data Unit
integer(kind=4) :: ijcod(3,gdf_maxdims) = 0    ! Axis names
integer(kind=4) :: pad_desc              ! For Odd gdf_maxdims only
!
!
! The first block length is thus
!   s_dim-1 + (2*mdim+4) + (4) + (8) +  (6*mdim+2) + (3*mdim+5)
! = s_dim-1 + mdim*(2+6+3) + (4+4+2+5+8)
! = s_dim-1 + 11*mdim + 23
! With mdim = 7, s_dim=11, this is 110 spaces
! With mdim = 8, s_dim=11, this is 121 spaces
! MDIM > 8 would NOT fit in one block...
!
! Block 2: Ancillary information
!
! The same logic of Length + Pointer is used there too, although the
! length are fixed. Note rounding to even number for the pointer offsets
! in order to preserve alignement...
!
integer(kind=4) :: posi_start  = 1
!
! POSITION
integer(kind=4) :: posi_words = 15    ! Position section length: 15 used + 1 padding
integer(kind=4) :: proj_start         !! = s_posi + 16     ! Pointer to next section
integer(kind=4) :: ijsou(3) = 0       ! 75  Source name
integer(kind=4) :: ijsys(3) = 0       ! 71  Coordinate System (moved from Description secti
real(kind=8) :: ra         = 0.d0     ! 78  Right Ascension
real(kind=8) :: dec        = 0.d0     ! 80  Declination
real(kind=8) :: lii        = 0.d0     ! 82  Galactic longitude
real(kind=8) :: bii        = 0.d0     ! 84           latitude
real(kind=4) :: epoc       = 0.0      ! 86  Epoch of coordinates
real(kind=4) :: pad_posi
!
! PROJECTION
integer(kind=4) :: proj_words = 9     ! Projection length: 9 used + 1 padding
integer(kind=4) :: spec_start !! = proj_start + 12
real(kind=8) :: a0      = 0.d0        ! 89  X of projection center
real(kind=8) :: d0      = 0.d0        ! 91  Y of projection center
real(kind=8) :: pang    = 0.d0        ! 93  Projection angle
integer(kind=4) :: ptyp = p_none      ! 88  Projection type (see p_... codes)
integer(kind=4) :: xaxi = 0           ! 95  X axis
integer(kind=4) :: yaxi = 0           ! 96  Y axis
integer(kind=4) :: pad_proj
!
! SPECTROSCOPY
integer(kind=4) :: spec_words  = 14   ! Spectroscopy length: 14 used
integer(kind=4) :: reso_start  !! = spec_words + 16
```

```
real(kind=8) :: fres       = 0.d0     !101  Frequency resolution
real(kind=8) :: fima       = 0.d0     !103  Image frequency
real(kind=8) :: freq       = 0.d0     !105  Rest Frequency
real(kind=4) :: vres       = 0.0      !107  Velocity resolution
real(kind=4) :: voff       = 0.0      !108  Velocity offset
real(kind=4) :: dopp       = 0.0      !     Doppler factor
integer(kind=4) :: faxi    = 0        !109  Frequency axis
integer(kind=4) :: ijlin(3) = 0       ! 98  Line name
integer(kind=4) :: vtyp    = vel_unk  ! Velocity type (see vel_... codes)
!
! RESOLUTION
integer(kind=4) :: reso_words = 3     ! Resolution length: 3 used + 1 padding
integer(kind=4) :: nois_start !! = reso_words + 6
real(kind=4) :: majo    = 0.0         !111  Major axis
real(kind=4) :: mino    = 0.0         !112  Minor axis
real(kind=4) :: posa    = 0.0         !113  Position angle
real(kind=4) :: pad_reso
!
! NOISE
integer(kind=4) :: nois_words = 2     ! Noise section length: 2 used
integer(kind=4) :: astr_start !! = s_nois + 4
real(kind=4) :: noise   = 0.0         ! 115 Theoretical noise
real(kind=4) :: rms     = 0.0         ! 116 Actual noise
!
! ASTROMETRY
integer(kind=4) :: astr_words = 3    ! Proper motion section length: 3 used + 1 padding
integer(kind=4) :: uvda_start !! = s_astr + 4
real(kind=4) :: mura     = 0.0        ! 118 along RA, in mas/yr
real(kind=4) :: mudec    = 0.0        ! 119 along Dec, in mas/yr
real(kind=4) :: parallax = 0.0        ! 120 in mas
real(kind=4) :: pad_astr
! real(kind=4) :: pepoch   = 2000.0     ! 121 in yrs ?
!
! UV_DATA information
integer(kind=4) :: uvda_words  = 18+2*code_uvt_last ! Length of section: 18 used + codes
integer(kind=4) :: void_start        !! = s_uvda + l_uvda + 2
integer(kind=4) :: version_uv = code_version_uvt_current  ! 1 version number.
integer(kind=4) :: nchan = 0         ! 2 Number of channels
integer(kind=8) :: nvisi = 0         ! 3-4 Independent of the transposition status
integer(kind=4) :: nstokes = 0       ! 5 Number of polarizations
integer(kind=4) :: natom = 0         ! 6. 3 for real, imaginary, weight. 1 for real.
real(kind=4)    :: basemin = 0.      ! 7 Minimum Baseline
real(kind=4)    :: basemax = 0.      ! 8 Maximum Baseline
integer(kind=4) :: fcol              ! 9 Column of first visibility information
integer(kind=4) :: lcol              ! 10 Column of last visibility information
! The number of information per channel can be obtained by
!       (lcol-fcol+1)/(nchan*natom)
```

```
    ! so this could allow to derive the number of Stokes parameters
    ! Leading data at start of each visibility contains specific information
    integer(kind=4) :: nlead = 7         ! 11 Number of leading informations
    ! Trailing data at end of each visibility may hold additional information
    integer(kind=4) :: ntrail = 0        ! 12 Number of trailing informations
    !
    ! Leading / Trailing information codes have been specified before
    integer(kind=4) :: column_pointer(code_uvt_last) = code_null ! Pointers to columns...
    integer(kind=4) :: column_size(code_uvt_last) = 0  ! Number of columns for each
    ! In the data, we instead have the codes for each column
    ! integer(kind=4) :: column_codes(nlead+ntrail)         ! Column code for each ...
    ! integer(kind=4) :: column_types(nlead+ntrail) /0,1,2/ ! Number of columns for each: 1 rea
    ! Leading / Trailing information codes
    !
    integer(kind=4) :: order = 0         ! 13  Stoke/Channel ordering
    integer(kind=4) :: nfreq = 0         ! 14  ! 0 or = nchan*nstokes
    integer(kind=4) :: atoms(4) = 0      ! 18  Atoms description
    !
    real(kind=8), pointer :: freqs(:) => null()     ! (nchan*nstokes) = 0d0
    integer(kind=4), pointer :: stokes(:) => null() ! (nchan*nstokes) or (nstokes) = code_stok
  end type gildas_header_v2
```

C UV tables column codes

```
! Column kind of the uv tables
integer(kind=4), parameter :: code_type_r4 = 1 ! Real number
integer(kind=4), parameter :: code_type_r8 = 2 ! Doubleprecision number
!
integer(kind=4), parameter :: code_follow   = -1 ! Column is a follower of the previous one
integer(kind=4), parameter :: code_uvt_u    =  1 ! u
integer(kind=4), parameter :: code_uvt_v    =  2 ! v
integer(kind=4), parameter :: code_uvt_w    =  3 ! w
integer(kind=4), parameter :: code_uvt_date =  4 ! Date
integer(kind=4), parameter :: code_uvt_time =  5 ! Time
integer(kind=4), parameter :: code_uvt_anti =  6 ! Antenna i
integer(kind=4), parameter :: code_uvt_antj =  7 ! Antenna j
integer(kind=4), parameter :: code_uvt_scan =  8 ! Scan number
integer(kind=4), parameter :: code_uvt_fobs =  9 ! Observatory frequency
integer(kind=4), parameter :: code_uvt_loff = 10 ! Phase center offset
integer(kind=4), parameter :: code_uvt_moff = 11 ! Phase center offset
integer(kind=4), parameter :: code_uvt_xoff = 12 ! X Pointing Offset
integer(kind=4), parameter :: code_uvt_yoff = 13 ! Y Pointing Offset
integer(kind=4), parameter :: code_uvt_stok = 14 ! Polarization state
integer(kind=4), parameter :: code_uvt_el   = 15 ! Elevation
integer(kind=4), parameter :: code_uvt_ha   = 16 ! Hour angle
integer(kind=4), parameter :: code_uvt_para = 17 ! Parallactic angle
integer(kind=4), parameter :: code_uvt_int  = 18 ! Integration time
integer(kind=4), parameter :: code_uvt_weig = 19 ! Weight column for SD
integer(kind=4), parameter :: code_uvt_xofi = 20 ! Pointing error Ant I
integer(kind=4), parameter :: code_uvt_yofi = 21 ! Pointing error Ant I
integer(kind=4), parameter :: code_uvt_xofj = 22 ! Pointing error Ant J
integer(kind=4), parameter :: code_uvt_yofj = 23 ! Pointing error Ant J
integer(kind=4), parameter :: code_uvt_ra   = 24 ! RA of reference
integer(kind=4), parameter :: code_uvt_dec  = 25 ! DEC of reference
integer(kind=4), parameter :: code_uvt_last = 25 ! Last uvt code
! currently up to 25 simultaneously present codes can fit in a single block
```

D GDF API and programming tools

D.1 GILDAS_NULL

The subroutine GILDAS_NULL is the principal initialization routine for GILDAS data structures. No access to the Gildas derived type elements should be done before calling this routine, as some elements are dangling pointers. Subroutine GILDAS_ERROR will return an error and issue a message if the gildas data structure has not been initialized. [5]

```
interface
  subroutine gildas_null(hx, type)
    use image_def
    !---------------------------------------------------------------------
    ! @ public
    ! GDF API
    !   Initialize a Gildas structure and reset its content
    !   TYPE can be: 'IMAGE' (default), 'TABLE', 'UVT', 'TUV', 'VOTABLE'
    !---------------------------------------------------------------------
    type(gildas), intent(out), target :: hx ! Gildas structure
    character(len=*), intent(in), optional :: type
  end subroutine gildas_null
end interface
```

It has an optional argument named type, which indicates which type of Gildas data structure is used. The default is 'IMAGE'.

Calling GILDAS_NULL(h, type = 'UVT') will set appropriate defaults for UV data set in natural order, while GILDAS_NULL(h, type = 'TUV') should be used for the transposed order. Existing pointers are nullified, and then associated to their appropriate target.

D.2 GDF_READ_HEADER

Subroutine GDF_READ_HEADER reads the header of the requested Gildas data set (specified in imag%file), and optionally change its rank.

```
subroutine gdf_read_header(imag,error,rank)
  use gio_dependencies_interfaces
  use gio_interfaces, except_this=>gdf_read_header
  use image_def
  use gbl_message
  !---------------------------------------------------------------------
  ! @ public
  ! GDF API
  !       Read an image header from the requested file
  !---------------------------------------------------------------------
  type(gildas), intent(inout) :: imag   ! Image structure
  logical,      intent(out)   :: error  ! Error flag
  integer,      intent(in), optional :: rank  ! Desired rank
```

[5] However, the whole GIO library can still work consistently on unitialized gildas data structure, as it never accesses the pointer elements.

If the optional `rank=value` argument is present, the header is modified according to the following rules. `value < 0` means the image must have `hx%gil%ndim=-rank`. `value = 0` means trim all possible trailine degenerate dimensions, i.e. set `hx%gil%ndim` to the last dimension > 1. `value > 0` means extend or trim rank to desired value.

D.3 GDF_TRIM_HEADER

Subroutine `GDF_TRIM_HEADER` changes the rank of an image header. See `GDF_READ_HEADER` for interpretation of the rank.

```
subroutine gdf_trim_header(imag,rank,error)
  use gio_dependencies_interfaces
  use gio_interfaces, only : gio_message
  use image_def
  use gbl_message
  !---------------------------------------------------------------------
  ! @ public
  ! GDF API
  !       Read an image header from the requested file
  !---------------------------------------------------------------------
  type(gildas), intent(inout) :: imag    ! Image structure
  integer(4),   intent(in)    :: rank    ! Requested Rank
  logical,      intent(out)   :: error   ! Error flag
```

D.4 GDF_ALLOCATE

The `gdf_allocate(header,error)` routine returns the appropriate Header%XnD pointer, depending on the data type and rank. It uses the `%blc, %trc` information to derive the proper size, given the data rank in `%gil%ndim`.

D.5 GDF_COPY_HEADER

Subroutine `GDF_COPY_HEADER` copy the information from one header to another. **The output header must have been initialized by a call to `GILDAS_NULL` before.**

```
interface
  subroutine gdf_copy_header(input,output, error)
    use image_def
    !---------------------------------------------------------------------
    ! @ public
    ! Copy (a part of) the input header into the output header.
    ! Since the arguments are the full gildas type, 'output' must be
    ! inout to avoid resetting the data part of the type. This implies that
    ! the other header components must also have been set/initialized
    ! before.
    !---------------------------------------------------------------------
    type (gildas), intent(inout), target :: input   ! Input header
    type (gildas), intent(inout), target :: output  ! Output header
    logical, intent(out) :: error
```

```
  end subroutine gdf_copy_header
end interface
```

The behaviour depends on the type of headers, defined as `h%gil%type_gdf`. If the destination header is of same type as the source header, which means the same *absolute value* of `h%gil%type_gdf`, all the relevant information is copied, including the source type, as this indicates the transposition status. If the destination header has a different type, only the common information is copied, so that the information which is only present in the destination header is preserved. This typically happens when copying an Image header to a UV header or vice-versa.

The routine may return an error. This is a change of interface compared to GILDAS V1.

D.6 GDF_RANGE

Function `gdf_range` converts an input range to a valid output one given a total number of items. It is a generic function, accepting any kind of integers.

```
interface gdf_range
  function gdf_range(nc,nchan)
    use gbl_message
    !---------------------------------------------------------------------
    ! @ public
    ! GDF API
    !       Return a range of (pixel / channel / items)
    !       Any Kind of integers is accepted
    !---------------------------------------------------------------------
    integer(4) :: gdf_range  ! Intent(out)  0 if no error
    integer(kind=*), intent(in) :: nchan     ! The actual number of channels
    integer(kind=*), intent(inout) :: nc(2)  ! The interpreted range of selected channels
  end function gdf_range
end interface
```

Positive values of `nc` indicate an absolute channel number, while negative values are offsets from the last channel. Null values default to 1 (first channel) and `nchan` (last channel) respectively.

D.7 GDF_READ_GILDAS

Subroutine `gdf_read_gildas` is the basic entry point to read a GILDAS data file into a GILDAS data structure. Based on the `hx%gil%type_gdf` code (previously set by an appropriate call to `gildas_null` or inherited by `gdf_copy_header` or `gdf_transpose_header`, it will perform different operations.

```
interface
  subroutine gdf_read_gildas(hx,name,ext,error, rank, data)
    use image_def
    use gbl_message
    !-----------------------------------------------------------------
    ! @ public
    ! Read a Gildas Data File of name "name" and extention "ext"
```

```
    ! and return it into the "hx" Gildas data type.
    !
    ! Desired Form & Rank can be specified: an error will happen
    ! in case of mismatch.
    !---------------------------------------------------------------
    type (gildas), intent(inout) :: hx    ! Gildas data type
    character(len=*), intent(in) :: name  ! Filename
    character(len=*), intent(in) :: ext   ! Extension
    logical, intent(out) :: error         ! Error flag
    integer, intent(in), optional :: rank
    logical, intent(in), optional :: data
  end subroutine gdf_read_gildas
end interface
```

Read a GILDAS data set and (optionally) return data from it.

Optional argument `data` is a logical indicating whether the data is read (`data=.true.`, which is the default) or only the header is read (`data=.false.`). The data is returned in the `hx%XNd` where `X = r, d` or `i` according to the data type, and `N = 1-4` is the data rank. The array is allocated by the routine.

Optional argument `rank=value` indicates the desired rank of the result. `value<0` means the image must have an intrinsic rank equal to `-value`. `value>0` means that the data must be trimmed of its last, or extended by, (degenerate) dimensions to match the requested rank; an error is returned if not possible. `value=0` means the header should take the rank from file, but the data should be returned as a 1D array.

Note: if preset on input, **hx%gil%form** indicates the desired type of values. An error will occur in case of mismatch: no conversion between real or integers is performed.

For UV data sets (in UVT or TUV order), it reads the header and the data according to the file layout.

D.8 GDF_READ_UVDATASET

`subroutine gdf_read_uvdataset(huvin,huvou,nc,duvou,error)`

Subroutine `gdf_read_uvdaset` is a somewhat more elaborate entry point which performs the possible layout conversion.

```
interface
  subroutine gdf_read_uvdataset(huvin,huvou,nc,duvou,error)
    use image_def
    use gio_image
    use gbl_message
    use gbl_format
    !-------------------------------------------------------------------
    ! @ public
    !       Read UV data and Associated parameters from a GILDAS UV
    !       structure and place it in the specified array
    !-------------------------------------------------------------------
    type(gildas), intent(inout) :: huvin     ! Input UV header (data file)
    type(gildas), intent(inout) :: huvou     ! Output UV header (program used)
```

```
    integer(kind=4), intent(in) :: nc(2)     ! Selected channels
    real(kind=4), intent(inout) :: duvou(huvou%gil%dim(1),huvou%gil%dim(2))  ! Data
    logical,      intent(out)   :: error     ! Flag
  end subroutine gdf_read_uvdataset
end interface
```

huvin must contain the data file header, while huvou can have a different layout. Layout can differ in several ways: different column_pointer and / or column_size arrays, transposition order (i.e. we may have huvou%gil%type_gdf = - huvin%gil%type_gdf). Transposition is done on-the-fly if needed. The two headers must otherwise be conforming.

nc(2) indicate the channel range to be retrieved. This is performed by calling function gdf_range(nc, huvin%gil%nchan) (or on a local copy, as nc is intent(in) only).

If the destination requires information not present in the source, or vice versa, an error is raised and a status code returned in huvou%status **TO BE CODED...**. It is up to the caller to react accordingly.

The routine currently assumes only one polarization. Polarization handling still needs further debate.

D.9 GDF_READ_UVONLY_CODES

```
subroutine gdf_read_uvonly_codes(huv,uv,codes,ncode,error)
  use gio_interfaces, except_this=>gdf_read_uvonly_codes
  use image_def
  use gio_image
  use gbl_message
  !---------------------------------------------------------------------
  ! @ public
  ! GDF API
  !       Read the Time-Baseline data of a GILDAS UV structure
  !       UVT and TUV order are allowed...
  !---------------------------------------------------------------------
  type (gildas), intent(inout) :: huv               ! UV table descriptor
  integer(4), intent(in) :: ncode                   ! Number of codes
  integer(4), intent(in) :: codes(ncode)            ! Desired codes
  real(kind=4), intent(out) :: uv(ncode,huv%gil%nvisi)  ! Return Values
  logical, intent(out) :: error                     ! Flag
```

Subroutine gdf_read_uvonly_codes allows to read only the columns containing the requested codes. On input, UVT and TUV order are supported.

For example, if the requested codes contains

$$code\_uvt\_u, code\_uvt\_v$$

, it would return the U and V values for all visibilities (in this example, gdf_read_uvonly would return the same information, but in a transposed way).

D.10 GDF_SETUV

Subroutine gdf_setuv verifies and finalizes a UV header. It computes the secondary parameters %nlead, %ntrail, %fcol and %lcol from the primary information present in %column_pointer,

%column_size, %nchan and %nstokes. It also verifies the conformance to the programming restrictions.

```
interface
  subroutine gdf_setuv (hx,error)
    use image_defgrep
    use gbl_message
    !---------------------------------------------------------------------
    ! @ public
    ! Define the UV section consistently.
    ! It could also define the column_types and column_codes if needed
    !---------------------------------------------------------------------
    type(gildas), intent(inout) :: hx
    logical, intent(out) :: error
  end subroutine gdf_setuv
end interface
```

D.11 Other less used or too specific or hidden routines... To be debated

```
interface
  subroutine gdf_get_baselines (mine,error)
    use image_def
    use gbl_message
    use gbl_format
    !---------------------------------------------------------------------
    ! @ public
    ! GDF API
    !   Compute the baseline range from a Virtual Memory UV data (one with the
    !   appropriate address field). Note that the address field
    !   can have been set independently of the %getvm status
    !---------------------------------------------------------------------
    type (gildas), intent(inout) :: mine   !
    logical,       intent(out)   :: error  !
  end subroutine gdf_get_baselines
end interface
```

Computes the min max of the baseline lengths. Used to update the header.

```
interface
  subroutine gdf_read_uvall (huv,array,error)
    use gio_interfaces
    use image_def
    use gio_image
    use gbl_message
    use gbl_format
    !---------------------------------------------------------------------
    ! @ no-interface (because of argument type mismatch)
    !       Read data from a GILDAS UV structure and
    !       place it in the specified array
```

```
    !----------------------------------------------------------------------
    type(gildas), intent(inout) :: huv       ! Image descriptor
    real(kind=4), intent(inout) :: array(*)  ! Data
    logical,      intent(out)   :: error     ! Flag
  end subroutine gdf_read_uvall
end interface
```

Read the full UV data set (all channels included). Implicitely used by `gdf_read_data` when possible.

```
interface
  subroutine gdf_write_uvall(huv,array,error)
    use gio_interfaces
    use image_def
    use gio_image
    use gbl_message
    !----------------------------------------------------------------------
    ! @ no-interface  (Yet...)
    !       Write UV data to an image file specified
    !       by its image structure. The image must have been
    !       opened for write before
    !----------------------------------------------------------------------
    type(gildas), intent(inout) :: huv        ! Image descriptor
    real, intent(inout) :: array(*)           ! Data
    logical, intent(out) :: error             ! Flag
  end subroutine gdf_write_uvall
end interface
```

Write the full UV data set (all channels included). Implicitely used by `gdf_write_data` when possible.

```
interface
  subroutine gdf_read_uvonly(huv,uv,error)
    use image_def
    use gio_image
    use gbl_message
    !----------------------------------------------------------------------
    ! @ public
    ! GDF API
    !       Read the UV data of a GILDAS UV structure
    !       UVT and TUV order are allowed...
    !----------------------------------------------------------------------
    type (gildas), intent(inout) :: huv               ! UV table descriptor
    real(kind=4), intent(out) :: uv(huv%gil%nvisi,2)  ! Return U,V coordinates
    logical, intent(out) :: error                     ! Flag
  end subroutine gdf_read_uvonly
end interface
```

Read only the U and V values.