Programming in GILDAS

Major revision on October 2008, but still not fully up-to-date

Questions? Comments? Bug reports? Mail to: `gildas@iram.fr`

The GILDAS team welcomes an acknowledgment in publications
using GILDAS software to reduce and/or analyze data.
Please use the following reference in your publications:
`http://www.iram.fr/IRAMFR/GILDAS`

Documentation

In charge: S. Guilloteau[1].
Active developers: S. Bardeau[2], J. Pety[2,3], E. Reynier[2].

Software (GILDAS kernel)

In charge: J. Pety[2,3].
Active developers: S. Bardeau[2], S. Guilloteau[1], E. Reynier[2].
Main past contributors: F. Badia, D.Broguière, G. Buisson, L. Desbats, G. Duvert, T. Forveille, R. Gras, P. Valiron.

1. Observatoire de Bordeaux
2. IRAM
3. Observatoire de Paris

Related information are available in

- GILDAS general introduction
- GREG: Graphical Possibilities
- SIC: Command Line Interpretor

Contents

Introduction

The GILDAS Programming Manual includes all necessary information to create applications based on the GILDAS tools. The tools includes:

- SIC the command line interpreter, written in FORTRAN and callable as a subroutine by any program.
- GREG the high level Graphic library.
- Independent *Tasks* for specific applications. The *Task* programming section describes how to create new GILDAS tasks.

The GTVIRT low level Graphic library is also described for completeness, although very few programs will actually require using its possibilities directly.

This manual contains several chapters. Chapter 2 (the SIC programming manual) is essential when constructing interactive applications. Chapter 3 (the GREG programming manual) is essential for all graphics applications. To create tasks, the user only needs to read Chapter 4 (the GILDAS task programming manual). Sophisticated graphic application may require to read Chapter 5, the GTVIRT programming manual.

Chapter 1

SIC Programming Manual - Partially updated on Oct.2008

1.1 Introduction

SIC (*) is a command line interpretor, written in FORTRAN and callable as a subroutine by any program. It provides a command language, with the following major features:

- resolution of command abbreviations
- definition of symbols
- macro capabilities with arguments substitution during execution
- log file
- multi-language structure
- loop buffers for repetitive actions
- variables, arithmetic and logical expressions evaluation
- structured logical tests
- error recovery
- stack buffer
- editing of command lines
- GUI interface on Motif, OS/X and MS-Windows systems
- optionally, a Python interface

This section concerns the programmer who wants to use SIC as the monitor of a simple or complex, evolutive, documented interactive program. It assumes that the reader is already familiar with all capabilities of SIC, so that he should be able to design a program (or better a system) around the SIC monitor.

It is in fact essentially a "CookBook" giving a list of recipes to build a program using SIC, or interface SIC with a preexisting ensemble of routines. Following these recipes should result in a complete success, that is, a program fulfilling the following requirements:

- Works in interactive or batch, reading commands on Standard Input This is called `Read Mode`.
- Can be called as a subroutine to execute a single command (`Execute Mode`). In case of error or if a `PAUSE` is encountered, the system switches to Read Mode for interactive error recovery or to read commands after the `PAUSE`.
- Is completed by a library version
- The library version can be used to make another language around it.
- Is portable on all Unix-like systems (including Mac OS/X) and MS-Windows.

The section 1.2 describes the command vocabulary structure, and the initialization sequence of SIC. Section 1.3 indicates how to write the corresponding `HELP` files. Section 1.4 describes how to retrieve arguments from a SIC command line. Section 1.6 indicates how to complete the program so that it can also be a library of routines. Finally, Section 1.7 indicates how to use SIC variables, and Section 1.9.1 give all entry points of the SIC monitor.

SIC programming interface is designed for the Fortran language. Interfacing with C language requires some system dependent precautions concerning argument passing mechanism, in particular for character strings.

1.2 Initializing SIC: Languages and Packages

SIC is a multi-package multi-language interpretor. A package is a collection of tools logically grouped together to provide some related applications, e.g. a plotting ensemble like GreG. A package can reference several languages: for example, GreG contains 4 languages: GREG1, GREG2, GREG3 for 1-D, 2-D and 3-D data plotting, and GTV for the basic plot actions. SIC itself has 5 languages: SIC, VECTOR, ADJUST, GUI and TASK, with the last two in "Library only" mode.

1.2.1 Packages

A main program using SIC is reduced to practically nothing: For example, the main program of `Mapping` is

```
program mapping
  !----------------------------------------------------------------------
  ! Main astro program
  !----------------------------------------------------------------------
  external :: mapping_pack_set
  !
  call gmaster_run(mapping_pack_set)
  !
end program mapping
```

All the job is done by `gmaster_run` which declares `Mapping` as the master program, and calls its package setup routine `mapping_pack_set`.

This package setup routine named `name_pack_set`, where `name` is the package name, contains e.g. for MAPPING

```
subroutine mapping_pack_set(pack)
  use gpack_def
  !
  type(gpack_info_t), intent(out) :: pack
  !
  external :: greg_pack_set
  external :: mapping_pack_init
  external :: mapping_pack_on_exit
  !
  pack%name='mapping'
  pack%ext = '.map'
  pack%depend(1:1) = (/ locwrd(greg_pack_set) /)
  pack%init=locwrd(mapping_pack_init)
  pack%on_exit=locwrd(mapping_pack_on_exit)
  pack%authors="J.Pety, N.Rodriguez-Fernandez, S.Guilloteau, F.Gueth"
  !
end subroutine mapping_pack_set
```

`pack` is here a Fortran derived-type variable, with the following elements

- `pack%name` The package name (Character(len=12))
- `pack%ext` The default extension for procedures (Character(len=12))
- `pack%depend` The addresses of the package setting routines required by this package (an integer array of type Address_Length, 32 or 64 bits depending on the machine). Note that a package may depend on several other ones. The dependent packages are automatically loaded.
- `pack%init` The address of the initialization routine (an integer of type Address_Length)
- `pack%init` The address of the cleaning on exit routine (an integer of type Address_Length)
- `pack%authors` A character string handling the authors name

The initialisation routine referred to by `pack%init` must setup all the package languages, and execute any code require to provide the initial setup of the package. In addition, it is recommended to setup the messaging facility: in the example below this is done using `name_message_set_id`.

```
subroutine mapping_pack_init(gpack_id,error)
  use sic_def
  !-----------------------------------------------------------------------
  !
  !-----------------------------------------------------------------------
  integer :: gpack_id
  logical :: error
  !
  ! Local language
  call init_clean
  !
  ! One time initialization
  call map_message_set_id(gpack_id)  ! Set library id
```

```
  !
  ! Language priorities
  call exec_program('SIC'//backslash//'SIC PRIORITY 1 CLEAN')
  !
  ! Specific initializations
  !
end subroutine mapping_pack_init
```

1.2.2 Language Definition

A language contains an ensemble of commands. Each language must be initialized by means of an appropriate call to the routine SIC_BEGIN. This routine has the following calling list:

```
SIC_BEGIN(LANG,HELP,MCOM,VOCAB,VERSION,Dispatch,Error)
```

`LANG` is a Character constant which gives the name of the language. `LANG` need not be distinct from the command names, and it must not include the character \ which will appear in the internal `HELP` of SIC. `MCOM` is the number of commands and options appearing in the language vocabulary `VOCAB`. All commands in a SIC language are `CHARACTER` constants in which the first character is a reserved code. Lower case characters are **not** allowed, but the special character "_" may appear. All other special characters have (or may have in the future) some specific meaning in SICThe following is the `DATA` initialization statement of SIC itself given as an example of language.

```
  integer :: sic_commands
  parameter (sic_commands=76)
  character(len=12) :: sic_vocab(sic_commands)
  data sic_vocab/                   &
    ' HELP',                        &  ! Must be the first one...
    ' @',                           &  ! Others by alphabetic order
    ' ACCEPT',  '/ARRAY','/BINARY','/COLUMN','/FORMAT','/LINE',        &
    ' BEGIN',                       &
    ' BREAK',                       &
    ' COMPUTE', '/BLANKING',        &
    ' CONTINUE',                    &
    ' DEFINE',  '/GLOBAL','/LIKE',  &
    ' DELETE',  '/SYMBOL','/VARIABLE','/FUNCTION',                    &
    ' EDIT',                        &
    ' ELSE',                        &
    ' EXAMINE', '/GLOBAL','/FUNCTION','/HEADER','/ADRESS','/ALIAS',    &
    '#EXIT',                        &
    ' END',                         &
    ' FOR',     '/WHILE',           &
    ' IF',                          &
    ' IMPORT',  '/DEBUG',           &
    ' LET',     '/NEW','/PROMPT','/WHERE','/RANGE','/CHOICE','/FILE',  &
                '/INDEX','/SEXAGESIMAL','/LOWER','/UPPER','/FORMAT',   &
                '/FORMULA','/REPLACE','/STATUS',                       &
```

```
' MESSAGE',                    &
' MFIT',    '/START','/STEP','/WEIGHTS','/EPSILON','/METHOD',      &
            '/ITERATIONS','/QUIET','/BOUNDS',                      &
' NEXT',                       &
' ON',                         &
' PAUSE',                      &
' PYTHON',  '/END',            &
'#QUIT',                       &
'#RECALL',                     &
' RETURN',                     &
' SAY',     '/FORMAT',         &
' SIC',                        &
' SORT',                       &
' SYMBOL',  '/INQUIRE',        &
' SYSTEM',                     &
' TYPE'/
```

The first character code is interpreted as follow

`<space>` Usual command

`/` Option of the preceding command

`#` Command forbidden in the stack and in procedures. Commands of this type are still written in the Log_File.

`$` Special command which must not be inserted automatically in the stack. Commands of this type are written in the Log_File

`*` Purely informative command, which is only useful in an interactive session. Commands of this type are not written in the Log_File and not inserted in the Stack.

The `*` code can also be used for complex commands using alphanumeric keywords that you wish to expand yourself before saving them in the Log_File and the Stack, or for commands that may require to output several records (See `SIC_INSERT, SIC_LOG` and `SIC_LIRE)`. The GREG command `DRAW` is an example of this.

The programmer should be careful about deciding what attribute to give to any command. All the options referring to a command must immediately follow it. At initialization time, SIC recognizes the options and set up a table of pointers connecting the options to their respective commands. A failure to respect the adequate order results in a very strange vocabulary.

`VERSION` is a character string which must contain the version number, and may contain the date of last modification, the name of the programmer,... The following example is recommended for optimum presentation (respect the alignment of the date for different languages):

```
VERSION='1.0   21-APR-1984   Programmer Name'
```

`Dispatch` is a subroutine to handle the dispatching of actions for all commands in the language.

`ErrorRoutine` is a subroutine to perform the error recovery action in case the execution of a command of this language returns an error. It may do nothing.

1.2.3 The command dispatching and handling

The dispatching routine is typically a big SELECT CASE based on the command name, like this one

```
subroutine run_clean (line,comm,error)
  use gbl_message
  !-----------------------------------------------------------------------
  ! CLEAN Main routine
  !     Call appropriate subroutine according to COMM
  !-----------------------------------------------------------------------
  character(len=*), intent(inout) :: line  ! Command line
  character(len=*), intent(in)    :: comm  ! Command name
  logical,          intent(out)   :: error ! Logical error flag
  !
  call map_message(seve%c,'CLEAN',line)
  !
  ! Analyze command
  select case (comm)
  case ('LOAD')
     call load_buffer(line,error)
  case ('READ')
     call read_image(line,error)
  case ('CLARK')
     call clark_clean(line,error)
  case ('FIT')
   ... etc...
  case default
     call map_message(seve%i,'CLEAN',comm//' not yet implemented')
  end select
  !
end subroutine run_clean
```

A handling routine for the a command looks like

```
      SUBROUTINE COM1(LINE,ERROR)
      CHARACTER*(*) LINE
      LOGICAL ERROR
      LOGICAL SIC_PRESENT
      INTEGER IARG1_OPT1
      REAL ARG1
!
! Test presence of option 1, and if so
!       Decode Argument 1 of this option with a default value
      IF (SIC_PRESENT(1,0)) THEN
          IARG1_OPT1 = 10
          CALL SIC_I4 (LINE,1,1,IARG1_OPT1,.FALSE.,ERROR)
          IF (ERROR) RETURN
          WRITE (6,*) 'Option 1 Set With Argument',IARG1_OPT1
```

```
      ENDIF
!
! Retrieves and decode first argument of the command
      CALL SIC_R4 (LINE,0,1,ARG1,.TRUE.,ERROR)
      IF (ERROR) RETURN
      WRITE (6,*) 'Command COM1 activated. ARG1',ARG1
! End of interface analysis, call a standard FORTRAN routine with
! all parameters now defined
      CALL SUB1(ARG1,ARG2,...,IARG1,...,ERROR)
      RETURN
      END
```

The command line buffer `LINE` must be passed by argument, never in a Module, to allow modularity and multi-language use.

1.3 The Help File

The `HELP` files are simple text files with two levels of help. The list of help topics in the library should be identical to the list of commands of the program. In addition, a specific topic named `LANGUAGE` should include a one line description of all commands, with a subtopic named `NEWS` which describe the latest news on the specific language.

The `HELP` files format is the following

```
1 TOPIC
    help for this topic
2 SUBTOPIC
    help for a subtopic of the previous topic
1 OTHER_TOPIC
    text for other_topic
1 ENDOFHELP
```

where 1 and 2 are in the first column of the text file, and followed by a single space. This format is easy to modify as it is a simple text file, but much slower to access because it is sequential. The `1 ENDOFHELP` (with no trailing characters) sequence indicates the end of the help file.

The help file name should be assigned to the logical name specified in the `SIC_BEGIN` call, or given explicitly.

1.4 Retrieving Arguments

To properly interface your program, you must know how to retrieve an argument from the command buffer processed by SIC. In standard SIC offers for each command up to 98 options, and for each option (or command) up to 98 arguments, the total number of arguments and options in a command line being limited to 99.

Arguments in SIC command lines are positional. The position of argument number `IARG` of option number `IOPT` of the current command is kept in SIC by means of internal pointers, By convention, argument number 0 refers to the option (or command) itself, and option number 0 to the command itself. Four standard routines are provided to decode the following type of arguments. All these routines do not modify the value of the argument if it is missing in the

command buffer. They may return an error condition and output an error message if requested in this case.

```
Argument Type   Routine Name
Integer*4         SIC_I4
Real*4            SIC_R4
Real*8            SIC_R8
Logical*4         SIC_L4
Character         SIC_CH
Character         SIC_KE          (Upcase converted keyword)
Any               SIC_DE          (Get variable descriptor)
```

The first four routines routines have the following calling list

```
          CALL SIC_name (LINE,IOPT,IARG,Arg,PRESENT,ERROR)
in which
          LINE        The command buffer
          IOPT        The option number (0=command)
          IARG        The argument number (0=command or option)
          Arg         The argument to be retrieved
          PRESENT     A flag indicating whether an error must occur if
                      the argument is not present in the command line
          ERROR       An error flag set in case of decoding errors,
                      or for missing argument when PRESENT is set.
```

SIC_CH and SIC_KE are slightly different, because they also return the length of the character string:

```
          CALL SIC_CH (LINE,IOPT,IARG,Arg,Larg,PRESENT,ERROR)
          CALL SIC_KE (LINE,IOPT,IARG,Arg,Larg,PRESENT,ERROR)
where
          Larg        Integer, true length of Arg
```

While SIC_CH returns any character string, with implicit formatting if necessary, but no case conversion, SIC_KE returns upcase keywords.

SIC_DE is similar to the other subroutines, except it returns a variable descriptor (sic_descriptor_t) instead of a Fortran scalar value. This gives the possibility for the user to pass to the command either an explicit value (*e.g.* 1.23, "Hello") or a SIC variable name of any kind and *any rank*, while the other subroutines above are not able to deal with non-scalar values. If the user has passed a variable name, its descriptor is returned. If he has passed an explicit value, a SIC scratch variable is created in memory: it must be destroyed at some point with a call to sic_volatile. The calling sequence is equivalent to the other subroutines:

```
          CALL SIC_DE (LINE,IOPT,IARG,DESC,PRESENT,ERROR)
where
          DESC is of type(sic_descriptor_t)
```

Two additional routines provide a way of testing the presence of an argument SIC_PRESENT(IOPT,IARG) returns the logical value .TRUE. if the required argument is present, and SIC_LEN(IOPT,IARG) returns the actual length of the argument (0 means the argument is

missing). A last routine, SIC_NARG(IOPT) indicates how many arguments are actually present for option IOPT.

Note that due to the structure of SIC vocabulary, IOPT is always the sequential number of the option as defined by the ordering in the DATA statement. Hence IOPT is not dependent on the order in which the options appear in the command buffer currently analysed.

SIC also includes some general character string manipulation routines which may be of interest in many other problems. All these routines are described more completely in section 1.9.1.

1.5 The messaging facility

This needs to be documented.

1.5.1 Writing on files: Fortran logical unit number

SIC has the following usage of Fortran logical unit numbers:

```
5        Input of Commands, shareable
6        Output of messages, shareable
50 to 99 Used by SIC_GETLUN
```

Units 5 and 6 are respectively accessed via usual READ and WRITE instructions.

Units 5 and 6 are respectively accessed via usual READ and WRITE instructions. If you wish to be able to reassign the input or/and output of your program to a file, you should also make your input and output in the same way (or use the SIC_WPR, SIC_WPRN routine). If you mix READ(5,format) and WRITE(6,format) instructions with PRINT, READ(*,format) or WRITE(*,format) instructions, you might experience some problem. Mixing with C output (printf routine for example) may yield to disordered output because of buffering. Use of routine GAGOUT , or of the messaging facility, is recommended to printout information.

The monitor uses additional logical units for the LOG file, for opening temporary files and for active procedures. These units are taken in the range 50 to 99. Units 50 to 99 are assumed to be available for the routine SIC_GETLUN . This subroutine returns the next available logical unit in this range, and keeps track of their usage. Conflict may occur if the user directly opens a file under a unit in this range without allocating the unit through a call to SIC_GETLUN. Other units are never used by SIC and other GILDAS programs.

1.5.2 Linking on Linux: TO BE UPDATED

On Unix system, the location of the GILDAS software environment is normally defined by executing a simple starting script, like this one, typically in the .bash_profile of the user.

```
export GAG_ROOT_DIR=/home/Guilloteau/gildas/gildas-exe-dev
export GAG_EXEC_SYSTEM=pc-cygwin-gfortran
source $GAG_ROOT_DIR/etc/bash_profile
```

The development environment can be accessed by typing

```
source $GAG_ROOT_DIR$/admin/gildas-env.sh
```

which selects the default compilers to build the system. The development environment architecture uses Makefiles, and the best to develop a new program is just to clone some architecture (see contrib/example for a full working example **To be Done**).

1.6 The Library Version

Once you have completed the previous operations, an interactive system is available. The system can also be used in batch or within command procedures. Each command handled interactively can also be executed from a program, using either `exec_command` (for simple commands), or `exec_program` (to execute a procedure, or complex commands calling such procedures).

```
call exec_command{command,error}
call exec_program(command)
```

where `command` is a character string containing the command to be executed. `exec_command` is reserved to execute atomic commands, and return an error in case of problems. On the other hand, `exec_program` is typically called with `command= '@ MyProcedure'`. It cannot return an error, but the procedure or code being executed must have its own error handling mechanism.

1.6.1 Library Only mode

It is sometimes a burden to have complex languages such as `GREG1\` in your `HELP` when you just want to use the library version of the language. SIC provides a way to load a language in "Library Only" mode. Commands of such languages are accessible only through the `exec_command` mechanism (or an equivalent coding). This mode is useful for building another more elaborate language from one (or more) lower level languages (pay attention to possible recursive programming in doing so however). This mode is called by loading the corresponding language with a negative size specification :

```
SIC_BEGIN(LANG,HELP,-MCOM,VOCAB,VERSION,Dispatch,ErrorRoutine)
```

A language loaded in such a mode can still be accessed interactively or in procedures: the command name must be given with the full language name in it to resolve the command. Similarly, the HELP can be accessed by giving the full language and command names.

1.7 Using Variables

SIC allows the definition and use of variables. Variables can be declared by the user, or by the program. In the latter case, SIC only remembers the address of the variable, examining its content only when required (command `EXAMINE` for example or any reference to the variable name in a command). This imposes two precautions in the FORTRAN code.

1. The attribute of the SIC variable (`READ-ONLY` or `READ-WRITE`) should be chosen carefully (e.g. a real-time application in which the time is declared as a SIC variable).

2. The corresponding FORTRAN variable must be `SAVE`d to prevent the compiler to allocate it on the stack.

1.7.1 Definition

There are 5 variable declaration routines following the same calling conventions

```
          Argument Data Type      Name of Subroutine
                REAL(KIND=8)            SIC_DEF_DBLE
                REAL(KIND=4)            SIC_DEF_REAL
                INTEGER(KIND=4)         SIC_DEF_INTE
                CHARACTER(LEN=*)        SIC_DEF_CHAR, SIC_DEF_STRN, SIC_DEF_CHARN
                LOGICAL(KIND=4)         SIC_DEF_LOGI, SIC_DEF_LOGIN

The calling syntax is the following
          CALL SIC_DEF_Name (NAME,VARIABLE,READONLY,ERROR)
for LOGI and CHAR, and
          CALL SIC_DEF_Name (NAME,VARIABLE,NDIM,DIM,READONLY,ERROR)
for DBLE, REAL, INTE, CHARN, and LOGIN,  NDIM being the number of dimensions,
and DIM the dimensions of the array VARIABLE, and
          CALL SIC_DEF_STRN (NAME,VARIABLE,LEN,READONLY,ERROR)
LEN being the length of VARIABLE.
```

`NAME` is the name of the SIC variable, `VARIABLE` the name of the corresponding FORTRAN variable, `READONLY` a logical indicating whether the variable should be Read-Only (`.TRUE.`) or Read-Write (`.FALSE.`). `ERROR` is a logical error flag set by SIC if the variable could not be defined.

In addition, `SIC_DEFSTRUCTURE(Name,readonly,error)` can be used to define SIC structures.

1.7.2 Assignment and Examination

SIC variables can be assigned new values, and examined using the following routines.

```
          Argument Data Type      Assignment    Examination Subroutine
          REAL(kind=8)            SIC_LET_DBLE   SIC_GET_DBLE
          REAL(KIND=4)            SIC_LET_REAL   SIC_GET_REAL
          INTEGER(KIND=4)         SIC_LET_INTE   SIC_GET_INTE
          LOGICAL(KIND=4)         SIC_LET_LOGI   SIC_GET_LOGI
          CHARACTER(LEN=*)        SIC_LET_CHAR   SIC_GET_CHAR

The calling syntax is the following
          CALL SIC_LET_Name (NAME,VARIABLE,ERROR)
          CALL SIC_GET_Name (NAME,VALUE,ERROR)
except for SIC_GET_CHAR
          CALL SIC_GET_CHAR (NAME,STRING,LENGTH,ERROR)
```

These a-priori useless routines can be used to modify variables defined in completely independent parts of a program such as the codes supporting two different SIC languages. For example the `GRAPHIC\` language interacts with `GREG1\` and `GREG2\` through SIC variables.

For assignment routines, the data type of the `VARIABLE` argument and of the SIC variable must match exactly. For retrieving routines, the data type to specify is the type of the `VALUE` argument; implicit conversion from the type of the SIC variable is done if possible, an error is returned otherwise.

Only `SCALAR` variables can be assigned or examined in this way. For `ARRAY` variables, you should use the subroutine `SIC_DESCRIPTOR`.

```
        CALL SIC_DESCRIPTOR (VARIABLE,DESC,FOUND)

  - VARIABLE (input) is the variable name (general syntax A[i,j]
    allowed)
  - DESC (output) is the descriptor of the variable, an instance of
    the derived type 'SIC_DESCRIPTOR_T' provided by the module
    'GKERNEL_TYPES', containing:
      DESC%TYPE     Variable type:
          * DESC%TYPE > 0 : Character string of length DESC%TYPE
          * DESC%TYPE < 0 may have the value
                  FMT_R4 for REAL(KIND=4) variable
                  FMT_R8 for REAL(KIND=8) variable
                  FMT_I4 for INTEGER(KIND=4) variable
                  FMT_I8 for INTEGER(KIND=8) variable
                  FMT_L  for LOGICAL(KIND=4) variable
                  FMT_BY for INTEGER(KIND=1) variable
                  FMT_C4 for COMPLEX(KIND=4) variable
      DESC%NDIM     Number of dimensions
      DESC%DIMS(7)  Dimensions
      DESC%ADDR     Variable address
      DESC%SIZE     Size of allocation (4-bytes words)
      DESC%STATUS   Origin of variable
                0       program defined
                < 0     user defined
                > 0     image
      DESC%READONLY True if the variable is read-only for the user.
   - FOUND (output) is a logical indicating whether the specified
     variable exists or not.
```

The parameters `FMT_R4, FMT_R8, FMT_I4, FMT_I8, FMT_L, FMT_BY` and `FMT_C4` are defined in the module `gbl_format`.

Three other routines may be required for variable handling:

```
        CALL SIC_MATERIALIZE(VARIABLE,DESC,FOUND)
```

is like SIC_DESCRIPTOR, but allow implicit transposition and sub-arrays in the variable name. An intermediate variable will be created to handle such cases.

`SIC_INCARNATE` is used to create an "incarnation" of a variable under a specified type `(REAL, DOUBLE, INTEGER)`.

```
        CALL SIC_INCARNATE(FORM,DESC,INCA,ERROR)
where
    - FORM is the format of the desired incarnation (FMT_R4, FMT_I4
      or FMT_R8)
    - DESC is the descriptor of the variable
    - INCA is the descriptor of the incarnation
    - ERROR is a logical error flag
```

DESC and INCA are instances of the derived type SIC_DESCRIPTOR_T. DESC should have been obtained by a previous call to SIC_DESCRIPTOR.

Once used, the materialization or incarnation may be deleted using routine SIC_VOLATILE

```
    CALL SIC_VOLATILE(INCA)
where
    - INCA is the descriptor of the incarnation/materialization.
```

1.7.3 Mathematical Formula Handling

Two subroutines are available to decode mathematical or logical expressions, SIC_MATH and SIC_LOGICAL. In addition, a subroutine is available to decode generalized sexagesimal notation, SIC_SEXA. The calling syntax is the following

```
CALL SIC_name (EXPRESSION,LENGTH,VARIABLE,ERROR)
```

where EXPRESSION is a character string containing the mathematical or logical expression to be evaluated, LENGTH is the number of characters of this expression, VARIABLE a REAL*8 (or LOGICAL*4) variable to receive the expression value, and ERROR a logical error flag. A generalized sexagesimal notation is for example A:B:C or A:B, where A, B and C may be variables or expressions or numbers. B and C values must of course be greater than or equal to 0 and less than 60. The value is returned in units of A, the leftmost field.

1.7.4 Deleting Variables

Declared variables can be deleted when no longer needed, using routine SIC_DELVARIABLE. The calling syntax is the following

```
CALL SIC_DELVARIABLE (NAME,USER,ERROR)
```

where NAME is the SIC variable name, USER a logical indicating whether program-defined variable are protected against deletion (.TRUE.) or can be deleted (.FALSE.), and ERROR a logical error flag.

1.8 Using Functions

It is possible to define user callable functions which are recognized in the mathematical formulae evaluations, using subroutine SIC_DEF_FUNC as follows

```
    CALL SIC_DEF_FUNC(NAME,SF,DF,NARG,ERROR,HLPFILE)
where
    NAME    is the function name
    SF      is the address of the single precision implementation of
            the function
    DF      is the address of the double precision implementation of
            the function
    NARG    is the number of arguments of the function.
    ERROR   is an error flag
    HLPFILE is a file name or SIC logical value where function help
            can be found (optional argument)
```

Both single and Double precision routines will be called like as (e.g. for two arguments)

```
    S = SF(arg1,arg2)      and      D = DF(arg1,arg2)
```

1.9 SIC Callable Routines

1.9.1 Monitor interface routines

```
Available subroutines:
        SIC_BEGIN       SIC_OPT         SIC_RUN
        SIC_INSERT      SIC_LOG         SIC_LANG
Available functions:
        SIC_CTRLC       SIC_LIRE        SIC_INTER_STATE
```

To build a program around the SIC monitor, different routines are provided. Some must always be used, like `SIC_BEGIN`, others are simply provided for additional capabilities. These routines have a standard `SIC_` like name and are fully described in this section.

`SIC_BEGIN (LANG,HELP,NCOM,CCOM,VERSION,Dispatch,Error)`

This subroutine initialize a SIC language and thus is usually the first one called. All arguments are unchanged by the routines and may be passed as immediate values.

- `LANG Character*(*).`
  The language name, as it will appear in the `HELP` and will be returned by SIC after command line processing. It is truncated to 12 characters if necessary.
- `HELP Character*(*).`
  The logical name for the `HELP` file corresponding to the language being initialized.
- `NCOM Integer.`
  The number of command and options, i.e. the dimension of the `CCOM` character array
- `CCOM Character*(*) array of dimension NCOM.`
  Contains the vocabulary of the language. The structure of the command vocabulary is described elsewhere in this document.
- `VERSION Character*(*).`
  A string to indicate the version number, the last modified date, the programmer name,... which will appear as a message with the language name at run time when using the SIC VERSION command
- `Dispatch` The dispatching routine
- `Error` The error handling routine

`SIC_INSERT(LINE)`

This routine is only useful when the Stack Buffer is used to store commands. This is the default, and the case when the flag `MEMORY` was set to `.TRUE.` in the first call to `SIC_OPT`. It can be changed with the `SIC MEMORY` command. `SIC_INSERT` is used to put a command line into the Stack buffer, a very useful possibility for some applications, when several lines must be put or when the command line must be expanded by the user program before the insertion is made.

Note however that command lines are automatically inserted in the Stack Buffer by the monitor for usual commands. This is the standard way of using the Stack insertion mode, since it automatically takes into account many things like the execution level before deciding whether

an insertion must occur. Using directly SIC_INSERT usually implies to disable the automatic insertion by the monitor (using the character code * in the definition of the associated command, c.f. Section 1.2). You need also to perform a call to SIC_LIRE to check the SIC execution mode (you should only call SIC_INSERT when SIC_LIRE() is equal to 0) and associated calls to SIC_LOG to write the same information in the LOG file. The insertion does not occur if the memory flag is turned off (SIC MEMORY OFF).

- LINE Character*(*).

SUBROUTINE SIC_LOG (LINE,NL,LIRE)

This subroutine writes LINE(1:NL) into the Log_File if LIRE=0. LIRE is here necessary for consistency with the internal pointer indicating if SIC is processing a macro, the stack or the loop. This pointer can be retrieved by the function SIC_LIRE. A call to SIC_LOG when LIRE is not 0 has no action.

- LINE Character*(*) Input
- NL Integer Input
- LIRE Integer Input

SUBROUTINE SIC_LANG(LANG)

This subroutine returns the name of the language corresponding to the last command analysed. It is called only when building the library version of a multi-language application.

- LANG Character*(*) is returned by the routine

LOGICAL FUNCTION SIC_CTRLC ( )

SIC itself traps the <^C> by generating a PAUSE at the end of the command which was being executed when <^C> was pressed. You may want in time-consuming applications to check yourself at specific points whether <^C> has been pressed. SIC_CTRLC allows you to do so, and resets an internal flag to .FALSE. when called. It returns .TRUE. if <^C> has been pressed since either the last command completed execution or the last time it was called (using the most recent event), .FALSE. otherwise.

INTEGER FUNCTION SIC_LIRE ( )

This subroutine returns the internal pointer of SIC indicating where SIC is currently reading its commands. SIC_LIRE may take the values

- -10 Subroutine mode
- -2 Reading in the Loop buffer
- -1 Called from SIC_RUN with ICODE = -1
- 0 Interactive mode
- I>0 Reading in macro number I

LOGICAL FUNCTION SIC_INTER_STATE( )

This subroutine returns .TRUE. if the session is interactive, .FALSE. otherwise. This is the case if SIC is ran during a batch process.

1.9.2 SIC Arguments Retrieving Routines

```
Available subroutines:
          SIC_L4          SIC_I4          SIC_R4          SIC_R8
          SIC_CH          SIC_KE          SIC_NEXT

Available functions:
          SIC_PRESENT     SIC_LEN         SIC_NARG        SIC_START
```

There are 5 retrieving routines following similar calling conventions

```
          Argument Data Type      Name of Subroutine
                REAL(KIND=8)            SIC_R8
                REAL(KIND=4)            SIC_R4
                INTEGER(KIND=4)         SIC_I4
                LOGICAL(KIND=4)         SIC_L4
                CHARACTER(LEN=*)        SIC_CH, SIC_KE

The calling syntax is the following
          CALL SIC_Name (LINE,IOPT,IARG,ARGUM,PRESENT,ERROR)
Except for SIC_CH and SIC_KE
          CALL SIC_CH   (LINE,IOPT,IARG,ARGUM,LARG,PRESENT,ERROR)
          CALL SIC_KE   (LINE,IOPT,IARG,ARGUM,LARG,PRESENT,ERROR)
```

- `LINE Character*(*)`
  is the last command line processed from which arguments are to be retrieved.

- `IOPT Integer`
  is the number of the option. 0 means the command. Options are numbered in the order where they appear in the vocabulary array after the command name.

- `IARG Integer`
  is the number of the argument of the option/command to be retrieved. 0 means the option/command itself.

- `ARGUM (Type depending on the subroutine called)`
  is the argument to be returned

- `LARG Integer`
  is the returned number of characters for `SIC_CH` and `SIC_KE`

- `PRESENT Logical`
  is a logical indicating whether the argument must be present `(.TRUE.)` or not `(.FALSE.)`.

- `ERROR An logical error flag`
  which is set if either i) `PRESENT` = `.TRUE.` and the argument is missing or ii) a decoding error occured.

SIC_KE is like SIC_CH, but expects a "keyword", and thus returns the argument after upper case conversion.

In addition there are other functions related to the arguments:

INTEGER FUNCTION SIC_NARG(IOPT)

returns the number of arguments present for option IOPT.

LOGICAL FUNCTION SIC_PRESENT(IOPT,IARG)

returns a logical value indicating the presence of the IARG-th argument of the IOPT-th option.

SUBROUTINE SIC_AMBIGS(FACILITY,NAME,FULL,VOCAB,MVOC,IVOC,ERROR)

search for NAME in the vocabulary VOCAB, and returns the corresponding pointer IVOC in the vocabulary and corresponding keyword FULL = VOCAB(IVOC) if NAME is a non-ambiguous abbreviation of FULL.

- FACILITY Character*(*)
  A character string containing the name of the calling subroutine. This string is printed before any failure message (ambiguous or non existing keyword).
- NAME Character*(*)
  The input name to be searched for in the vocabulary
- FULL Character*(*)
  The complete keyword returned.
- IVOC Integer
  The keyword number in the vocabulary.
- MVOC Integer
  The number of keywords in the vocabulary.
- VOCAB Character*(*)(MVOC)
  The vocabulary.
- ERROR Logical
  An error flag set if the input name was ambiguous, invalid or not found in the vocabulary.

Typically, NAME will have been recovered by a call to SIC_KE

INTEGER FUNCTION SIC_LEN(IOPT,IARG)

returns the total length of the IARG-th argument of the IOPT-th option. Returns 0 if SIC_PRESENT(IOPT,IARG)=.FALSE.

INTEGER FUNCTION SIC_START(IOPT,IARG)

returns a pointer to the address of the argument in the character string. Returns 0 if SIC_PRESENT(IOPT,IARG)=.FALSE.. This routine can be used with SIC_LEN for special processing of arguments, such as sexagesimal decoding.

SUBROUTINE SIC_NEXT(LINE(NEXT:),ARGUM,LARG,NEXT)

This subroutine can be used for special processing of text with a syntax similar to command lines, i.e. text in which the delimiters are single spaces, strings being included between double quotes. Provided it is called with the text restricted to the current delimiter position LINE(NEXT:), it returns the position of the next delimiter NEXT (space, skipping strings), the character string included between these delimiters, ARGUM and its length LARG. This string may further be decoded as double precision value using routine SIC_MATH or SIC_SEXA, logical value using routine SIC_LOGICAL, or used as character constant.

1.9.3 Command Line Interpretor Subroutines

SUBROUTINE SIC_ANALYSE (COMMAND,LINE,NLINE,ERROR)

This subroutine analyses the command line LINE, set up all the internal pointers for argument retrieval, and returns the name of the command found. The line must have been formatted by SIC_FORMAT before being analysed. All active languages are considered, even the "Library Only" languages

- COMMAND Character(LEN=*), Command name (returned)
- LINE Character(LEN=*), Line to be analysed. It is modified by SIC_ANALYSE
- NLINE Integer. Current length of LINE, modified on return.
- ERROR Logical. Return error code

SUBROUTINE SIC_FORMAT(LINE,NLINE)

This subroutine does an adequate formatting of any line for later processing by the SIC interpretor. It does not use any internal common of SIC and thus may be accessed independently and considered as a more general routine.

- LINE Character*(*), Line to be formatted.
- NLINE Integer, Current length of LINE. It is modified by SIC_FORMAT and must be initialized so that only trailing spaces appear in LINE after NLINE. A good policy is to initialize NLINE to LENC(LINE).

1.9.4 All Purpose General Subroutines

```
Available subroutines:
          SIC_LOWER       SIC_PARSEF      SIC_UPPER       SIC_WPR
          SIC_WPRN
Available functions (Obsolescent)
          SIC_GETVM       FREE_VM         LENC
```

These routines are of more general use than SIC itself.

SUBROUTINE SIC_LOWER(LINE) - SIC_UPPER(LINE)

Converts the string LINE to lower or upper case letters respectively.

SUBROUTINE SIC_PARSEF (NAME,FILE,DEF,EXT)

Parses the file name FILE for a default directory DEF and extension type EXT. These item are added to FILE if needed, and NAME returns the short NAME of the file, that is, the name without extension or directory. All arguments are Character*(*). This is the standard way to obtain a file name in the GILDAS environment: this routine takes care of logical names (in the SIC meaning).

SUBROUTINE SIC_WPR(PROMPT,LINE)

Reads the string LINE with the prompt PROMPT. It tries to obtain a non-blank string, prompting again if it reads a blank string. If it receives a <^Z> code, it returns the string "EXIT". The prompt is automatically disabled if the session is not interactive.

- PROMPT Character*(*), Unchanged by SIC_WPR
- LINE Character*(*), Returned line.

`SUBROUTINE SIC_WPRN(PROMPT,LINE,N)`

Reads the string `LINE` with the prompt `PROMPT`. Returns N the number of characters read. Returns N=0 if it reads a blank line, or receives a `<^Z>` code. The prompt is automatically disabled if the session is not interactive.

- `PROMPT` Character*(*), Unchanged by `SIC_WPRN`
- `LINE` Character*(*), Returned line.
- `N` Integer, Number of characters in `LINE`

1.9.5 Symbol Manipulation Routines

Three routines are available to define, translate and delete symbols within a program using SIC without using the `SYMBOL` command.

`SUBROUTINE SIC_SETSYMBOL(SYMBOL,TRANSLATION,ERROR)`

Defines a symbol of name `SYMBOL` (Character*(*), maximum 12 characters), translation `TRANSLATION` (Character*(*), maximum 132 characters). An error flag (`ERROR` logical) is returned if the symbol could not be defined. Any previous definition is overridden.

`SUBROUTINE SIC_GETSYMBOL(SYMBOL,TRANSLATION,ERROR)`

Obtains the current translation of a given symbol. If `TRANSLATION` is too short, the translation is truncated without warning or error. `ERROR` is returned if the symbol is undefined only.

`SUBROUTINE SIC_DELSYMBOL(SYMBOL,ERROR)`

Deletes a symbol definition. `ERROR` is returned if the symbol was not defined.

1.10 Obsolescent Features

1.10.1 Function

`INTEGER FUNCTION LENC(LINE)`

Returns the "current" length of the string `LINE`, that is the position of the last significant character. It thus allows to ignore trailing blanks. It is obsolescent, as the Fortran-90 intrinsic LEN_TRIM provides the same functionality.

`INTEGER FUNCTION SIC_GETVM(N32,ADDR)`

`SIC_GETVM` allocates virtual memory for `N32` 32-bit words, and returns the allocated address space into the `KIND=ADDRESS_LENGTH` variable `ADDR`. `SIC_GETVM` is set to 1 in case of success, to other values otherwise.

`SUBROUTINE FREE_VM(N32,ADDR)`

`FREE_VM` can be used to free the corresponding adress space when no longer needed.

`SIC_GETVM` and `FREE_VM` use is being deprecated, as the ALLOCATE/DEALLOCATE mechanism of Fortran-90 offers an equivalent functionality.

1.10.2 Routines for Library usage (Obsolescent, for record only)

`SIC_OPT (PROMPT,LOGFILE,MEMORY)`

This routine is not compulsory. It is used to set the prompt the log file name and the stack usage. This routine only changes the prompt if called after `SIC_BEGIN`.

- `PROMPT Character*(*).`
  The prompt to be used in interactive mode, truncated to 8 characters if necessary. Default is `SIC.` Note that the caret `>` and other alterations such as : or `_1>` are added by SIC at run time and should not be included.

- `LOGFILE Character*(*).`
  The log file name. Default is `LOG`.

- `MEMORY Logical.`
  Indicates whether the Stack is used or not.

`SIC_RUN (LINE,LANG,COMMAND,ERROR,ICODE,OCODE)`

This subroutine is used to enter SIC, retrieve and analyse a command line for further execution. All SIC possibilities, including execution levels and error recoveries, made accessible (In particular, the `LINE` command line can be a call to a macro file).

- `LINE Character*(*) (maximum size is 2048)`
  Command line to be executed. Modified by SICNo need to initialize it when ICODE = 0 or 1, but must be initialized is ICODE = -1 or 2

- `LANG Character*12`
  Name of the language returned by SIC

- `COMMAND Character*12`
  Name of the command returned by SIC

- `ERROR Logical`
  Return error code

- `ICODE Operation code`

  - ICODE = -1
    Analyse the command line passed as argument LINE, and return to calling program to execute it.
  - ICODE = 0
    Loop into SIC to retrieve a new command line LINE, and return to calling program to execute it.
  - ICODE = 1
    Start SIC, and retrieve a first command line.
  - ICODE = 2
    Start SIC, analyse the command line passed as argument, and return to calling program to execute it.

- `OCODE Return code`

  - OCODE = -1
    End of execution in sub-routine mode. Program must return immediately to its caller.

- OCODE = 0
  Successful analysis of a command. Program must execute it, and then loop again on `SIC_RUN` with ICODE=0 to get further commands.
- OCODE = 1
  End of execution caused by typing the EXIT command. Program should perform any necessary action (close files, etc...) and return to its caller.

Chapter 2

GreG Programming Manual

Partially updated in Oct 2008. Need more details about loading GreG.

This section describes how GREG normally used as an interactive plot utility, can also be used as a high level plot library. GREG can be used exactly as a standard graphic library, but because of the possibilities of the command line monitor, many other possibilities are accessible.

Before presenting in detail the "Library Version" of GREG we should distinguish between three different possible applications of the GREG Library :

1. The occasional user who has a single repetitive graphic problem which is part of another complex program. Interactive control is not wanted. This case can often be solved using command procedures and SIC images as data format when formatted I/O is definitely too slow. This is very efficient and flexible. If the user already discarded this possibility, he (she) most likely wants the simplest programming ever possible, will be satisfied by standard default values, and is not really worried about optimum efficiency.

2. A programmer wanting to solve a single repetitive graphic problem for use by other people. Interactive action is not wanted. Simple programming is of little importance, but efficiency is a major problem.

3. A programmer wishing to integrate elaborate and flexible graphic applications as part of a more complete data analysis system. Interactive control by the user and error recovery are necessary. Then all GREG capabilities are wanted, and in addition this programmer may well be interested to use the possibilities of the SIC monitor to "supervise" the data analysis system.

The library version of GREG allows all three cases to be solved adequately by offering three different ways to call GREG services :

1. Passing a command to GREG using the routine `GR_EXEC`
   `GR_EXEC('Command Argument/Option')` will execute the command line exactly as if you had typed it interactively. Instead of `GR_EXEC`, it is recommended to use `GR_EXEC1`, `GR_EXEC2` and `GR_EXECL` for commands of languages GREG1, GREG2 and GTVL respectively.

2. Calling an intermediate formatting routine which generates the appropriate call to `GR_EXEC` from its own arguments. Special entries are used to process possible options. The advantage of this mode is to provide a more standard program interface. Not all commands will be accessible in that way, and it is marginally slower than the previous mode.

3. Calling subroutines which do not correspond to GREG commands but directly generates plot actions. This way is the most optimized access, since it bypasses the command line interpreter. However, only standard things can be done like this, and it requires some precautions because of the segmented nature of the graphic library.

All three ways can be used within a single program, and the choice between one or the other is just a matter of convenience and/or efficiency.

In addition, the GREG plot library can be used either as a classic package of subroutines ("Library Only"), or as an interactive facility allowing user control at run time by means of the SIC monitor possibilities ("Interactive"). The two modes can be mixed in a subroutine, with the important restriction that the "Library Only" mode is just a subset of the "Interactive" mode. It is not possible to change the mode during program execution. A program using GREG plot library in "Interactive" mode in fact appears as a superset of the GREG interactive utility.

2.1 Interaction with FORTRAN programs

For a normal user, it can be said that GREG does not interfere at all with a `FORTRAN` program. All interactions with a program concern the system or SIC facilities as detailed below.

- I/O Units :
  All logical units used by the GREG system and the associated SIC monitor are FORTRAN units between 50 and 99. Programs using SIC should avoid opening such units, or should get available units through calls to `SIC_GETLUN`.

- Work Space :
  GREG uses as much as possible the concept of virtual memory. This means that work space, when required, is allocated dynamically at run time. Hence, GREG does not overload a small program. There is currently one exception, the X,Y and Z buffers which have a fixed size allocation of 10000 long-words each. This may change at some time. On large applications, be sure that your virtual memory quota is large enough. Be sure also to run GREG in a large enough working set to reduce page faults.

- Special Handler :
  The SIC monitor always traps the `^C` action to provide a facility to interrupt procedures at any time. You can bypass this action by adequate programming (see `SIC_CTRLC` routine in the SIC manual).

Programmers using SIC as command monitor together with GREG either in interactive or in library mode, should be aware of the interaction between GREG and SIC command parsing facility. Each call to `GR_EXEC, GR_EXEC1, GR_EXEC2, GR_EXECL or EXEC_GREG` parses at least one GREG command line, and thus modify the pointers accessed by SIC argument retrieving routines. Accordingly, any subroutine implementing a user command should retrieve all its arguments **before** calling a GREG subroutine.

2.2 Basic Routines

1. `LOAD_GREG (Mode)`
   To be updated since the change of SIC initialization mode This subroutine must

be called before any other reference to GREG services. It is used to define the operating mode and to initialize the GREG vocabulary into the SIC monitor. The argument is of type Character and can be `'LIBRARY'` to initialize GREG in the "Library Only" (no interactive SIC monitor) mode, or `'INTERACTIVE'` to initialize GREG with all SIC monitor capabilities. Strings like `'LIBRARY GREG1'` or `'INTERACTIVE GREG2'` can be used to load only one language, although the GREG2\ language requires GREG1\ to function. This routine does not set the SIC prompt, neither the Log File which you may specify by calling routine `SIC_OPT`. Please refer to the SIC manual.

2. `GR_EXEC(LINE)`
   This is the basic routine for all plot actions. It is able to activate any GREG command exactly in the same way as if you had typed it during an interactive session. Use `GR_EXEC` if you do not know to which of the GREG languages it pertains, `GR_EXEC1,2 or L` according to the language if you know (and you should). The command line must not include the language field. The command is not written to the SIC stack, neither to the Log File.

3. `GR_ERROR()`
   This logical function allows error recovery. It returns the internal error status of GREG program, and clears it. If another GREG subroutine is called while an error status exist, program execution aborts.

4. `EXEC_GREG(LINE)`
   Obsolescent, use EXEC_COMMAND or EXEC_PROGRAM instead execute a command line which can be either a GREG command or a SIC command (like `'@ PROC'` for example). Control returns to the calling program when the command is completed successfully. If an error occurs, the current SIC error recovery action is activated before. This is usually a `PAUSE` which gives interactive control to the user ; control will then return to the calling program only after command `EXIT` (or `^Z`) has been typed. Contrary to `GR_EXEC`, the command is written to the SIC stack, and to the Log File. "Interactive" mode is required.

Only `GR_EXEC` *(or its variant) and* `GR_ERROR` *are allowed in Library Mode.*

2.3 Linking

2.3.1 UNIX systems

To be updated with current information on architecture under Linux

To access GREG from your Fortran Program, you need to link to several libraries. All GREG libraries are located in `$GAG_ROOT/lib` where `$GAG_ROOT` is an environment variable defining where the GILDAS software is located. See your local GILDAS expert for that.

Then you should use the following link command to link your program

```
f77 -o Program Program.f -L/usr/lib/X11 -L$GAG_ROOT/lib \
-lgdf -lGREG -lcontour -lgtlang -lchar -lsic -limage -lgag -lrary \
-lX11 -lm -lc
```

where

```
f77              is the Fortran compiler/linker
```

```
Program.f        is your Fortran program.
/usr/lib/X11     indicates where the X11 libraries are located
-lm -lc          are required on some systems
```

On some systems, the GREG libraries are shareable, thus avoiding code to be included in your own programs.

2.4 Running

If you are using GREG in "Library Only" mode, you can just run your program normally.

If you are using the "Interactive" mode you need to provide the assignments of GREG help files and working files to your program. All compulsory assignments are made system wide, so you can just run the program. You will obtain help for the SIC command monitor and a log file will be written in `.gag/logs` .

2.5 Example

No simple working example is available as of today. See the MAPPING program for a more exhaustive example.

2.6 Array Transfer

Subroutines with data arrays transferred to or obtained from GREG have a Real*4 and a Real*8 version. The simple precision has a name beginning by `GR4_`, the double precision a name beginning by `GR8_`. In the argument list description, `REAL` will mean Real(KIND=4) for the `GR4_` version, Real(KIND=8) for the `GR8_`. These routine work by copying the data. For really big arrays, it is more efficient to transfer the information by reference, though SIC variables (see the SIC manual).

2.6.1 GR4_GIVE - GR8_GIVE

`GR4_GIVE(NAME,NXY,ARRAY)`

This subroutine passes One-Dimensional array `ARRAY` to GREG as the X, Y, or Z array according to the given `NAME`. It is the optimal way to initialize the One-Dimensional arrays of GREG with the data you have computed within your application program.

```
NAME    is CHARACTER(LEN=1) and may be X, Y or Z
NXY     is the number of values set in ARRAY
ARRAY   is a REAL array of size NXY.
```

2.6.2 GR4_GET - GR8_GET

`GR4_GET(NAME,NXY,ARRAY)`

This subroutine is the reverse of the `GR4(8)_GIVE` routines. It passes One-Dimensional array `ARRAY` from GREG to your program as the X, Y, or Z array according to the given `NAME`. It allows to benefit in your program of the flexible input formats of GREG .

```
NAME    is CHARACTER(LEN=1) and may be X, Y or Z     Input
NXY     dimension of ARRAY                          Input
        number of values returned in ARRAY          Output
ARRAY   is a REAL array of size NXY                 Output
```

2.6.3 GR4_RGIVE - GR8_RGIVE

GR4_RGIVE(NX,NY,CONV,R)

This subroutine passes a two-Dimensional array to GREG as the Regular Grid array used for mapping.

```
NX,NY   Integers, array dimensions
CONV    Real(KIND=8) array of dimension 6
        It contains respectively
        CONV(1) Reference pixel in X (First dimension of R)
        CONV(2) X User coordinate value at CONV(1)
        CONV(3) X User coordinate increment per pixel (may be negative)
        CONV(4)-CONV(6) same as above for Y (Second dimension of R)
R       REAL   array (NX,NY) to be transferred.
```

If R is Real(KIND=4), the array is not physically copied but its address is computed by GREG for later use. GREG will not modify anything in it. If of type Real(KIND=8), virtual memory is allocated to create a Real(KIND=4) array of same dimensions.

2.6.4 GR4_LEVELS - GR8_LEVELS

GR4_LEVELS(NL,LEVELS)

This subroutine initializes a set of contour levels for mapping by GREG . LEVELS is a REAL array of dimension NL.

2.7 Immediate Routines

The following routines do not use the SIC command line interpretor. They are essentially internal GREG routines made available to the user, and are thus very efficient to use. As above, they have a KIND=4 and a KIND=8 version.

For optimization purposes, these routines do not include any explicit segmentation of the plot. In order to be consistent with the philosophy of the interactions between GREG and the GTVIRT graphic library, you must enclose a set of calls to the immediate routines between a call to GR_SEGM and GR_OUT. Ex:

```
CALL GR_SEGM('Nom',ERROR)  ! Close current graphic segment and open
                           ! a new one
CALL GR4_CONNECT (...)     ! Fill this segment with plot coordinates
CALL GR4_MARKER (...)      ! etc..
CALL GR_OUT                ! Make the segment visible on screen
```

2.7.1 GR_SEGM

GR_SEGM(NAME,ERROR

Close the current graphic segment and open a new graphic segment with the current plotting attributes selected by PENCIL. All the commands of GREG open at least one new segment, but the immediate routines do not. This routine must be called prior to calling a series of immediate routines to ensure that the plot will appear with the currently selected graphic attributes, and to allow a selective erasure of the plot. NAME is a character string indicating the desired name of the segment, and ERROR a logical error flag. All plot request issued between two successive calls to GR_SEGM will make a single graphic segment. Note however that every GREG command having an effective plot action creates one or more graphic segments.

2.7.2 GR_OUT

GR_OUT

Updates the graphic output. This routine must be called when you wish to make visible a series of calls to the immediate routines.

2.7.3 DRAW - RELOCATE

DRAW(XU,YU) RELOCATE(XU,YU)

Basic pen down or up movement in User coordinates (Real(KIND=8) values). There is no Real(KIND=4) version.

2.7.4 GDRAW - GRELOCATE

GDRAW(X4,Y4) GRELOCATE(X4,Y4)

Basic pen down or up movement in Physical coordinates (Real(KIND=4) values). There is no Real(KIND=8) version.

2.7.5 GR4_PHYS_USER - GR8_PHYS_USER

GR4_PHYS_USER (XP,YP,XU,YU,NXY)

Convert Physical coordinates (XP,YP always Real(KIND=4) values) to User coordinates (XU,YU, REAL values). NXY is the number of values.

2.7.6 GR4_USER_PHYS - GR8_USER_PHYS

GR4_USER_PHYS(XU,YU,XP,YP,NXY)

Convert User coordinates (XU,YU, REAL values) to Physical coordinates (XP,YP, always Real(KIND=4) values). NXY is the number of values.

2.7.7 GR4_CONNECT - GR8_CONNECT

GR4_CONNECT(NXY,X,Y,BVAL,EVAL)

This subroutine connects all data points represented by the X and Y arrays passed in arguments. BVAL and EVAL are used for blanked values. EVAL negative means no blanking.

```
Arguments :
        NXY     INTEGER         Input
        X       REAL (NXY)      Input
        Y       REAL (NXY)      Input
        BVAL    REAL            Input
        EVAL    REAL            Input
```

2.7.8 GR4_HISTO - GR8_HISTO

GR4_HISTO(NXY,X,Y,BVAL,EVAL)

```
Arguments :
        NXY     INTEGER         Input
        X       REAL (NXY)      Input
        Y       REAL (NXY)      Input
        BVAL    REAL            Input
        EVAL    REAL            Input
```

2.7.9 GR4_MARKER - GR8_MARKER

GR4_MARKER(NXY,X,Y,BVAL,EVAL)

Markers of current size are plotted at each data point.

```
Arguments :
        NXY     INTEGER         Input
        X       REAL (NXY)      Input
        Y       REAL (NXY)      Input
        BVAL    REAL            Input
        EVAL    REAL            Input
```

2.7.10 GR4_CURVE - GR8_CURVE

GR4_CURVE(NXY,X,Y,Z,VAR,PER,BVAL,EVAL,ERROR)

Plots a smooth curve from the (X,Y) values using the requested interpolant. Z is either a dummy argument or a parameter for the curve representation depending on the VAR value. VAR indicates which is the variable to use for interpolation. PER indicates whether the curve is periodic or not. BVAL and EVAL define the blanking value and the blanking tolerance (set EVAL negative to disable blanking checking). ERROR is an error flag set if the curve could not be produced. The current accuracy is used for the interpolation.

```
Arguments :
        NXY     INTEGER         Input
        X       REAL (NXY)      Input
        Y       REAL (NXY)      Input
        VAR     CHARACTER*(*)   Input
        PER     LOGICAL         Input
        BVAL    REAL            Input
        EVAL    REAL            Input
        ERROR   LOGICAL         Output
```

2.7.11 GR4_EXTREMA - GR8_EXTREMA

GR4_EXTREMA(NXY,X,BVAL,EVAL,XMIN,XMAX,NMIN,NMAX)

Compute the extrema of the input array avoiding blanked values.

```
Arguments :
        NXY     INTEGER         Input   Number of points
        X       REAL (NXY)      Input   Array
```

```
        BVAL    REAL            Input   Blanking value
        EVAL    REAL            Input   Tolerance on blanking
        XMIN    REAL            Output  Minimum value
        XMAX    REAL            Output  Maximum value
        NMIN    INTEGER         Output  Pixel of the minimum X(NMIN) = XMIN
        NMAX    INTEGER         Output  Pixel of the maximum
```

2.7.12 GR8_BLANKING

`GR8_BLANKING(BVAL,EVAL)`

Define the blanking value to be used later. It is equivalent to call `GR_EXEC('SET BLANKING BVAL EVAL')`, but the later form requires formatting of values.

```
Arguments :
        BVAL    REAL            Input   Blanking value
        EVAL    REAL            Input   Blanking precision
```

2.7.13 GR8_SYSTEM - GR8_PROJEC

`GR8_SYSTEM (ICODE)`

`GR8_PROJEC (X,Y,A,ICODE)`

Define respectively the coordinate projection system and the projection constants. The `SYSTEM` code can be 1 for `UNKNOWN`, 2 for `EQUATORIAL (1950.0)`, 3 for `GALACTIC`. The projection code is 0 for `NONE`, 1 for `GNOMONIC`, 2 for `ORTHOGRAPHIC`, 3 for `AZIMUTHAL`, 4 for `STEREOGRAPHIC`, 5 for `LAMBERT` cylindrical, 6 for `AITOFF` equal area and 7 for `RADIO` (also known as global SINUSOIDAL) projection.

```
Arguments :
        ICODE   INTEGER         Input
        X       REAL            Input       Projection center
        Y       REAL            Input       Projection center
        A       REAL            Input       Projection Angle
```

2.7.14 GR4_RVAL

`GR4_RVAL(XU,YU,Z4)`

Returns a map value at a given point in user coordinates.

```
Arguments :
        XU      REAL*8          Input
        YU      REAL*8          Input
        Z4      REAL*4          Output
```

2.7.15 GR_WHERE

`GR_WHERE(XU,YU,X4,Y4)`

Returns the pen position with the same conventions as `GR_CURS`

```
Arguments :
        XU      R*8     X User coordinates          Output
        YU      R*8     Y User coordinates          Output
        X4      R*4     X Plot coordinates          Output
        Y4      R*4     Y Plot coordinates          Output
```

2.7.16 GR8_TRI

`GR8_TRI(X,INDEX,N,*)`

Sorting program that uses a quicksort algorithm. Applies for an input array of Real(KIND=8) values which is reordered. It also returns an array of indexes sorted for increasing order of X. You can use `GR8_SORT` to reorder other arrays associated with X.

```
Arguments :
        X       R*8(*)  Unsorted/Sorted array           Input/Output
        INDEX   I(*)    Integer array of sorted indexes Output
        N       I       Length of arrays                Input
        *       Label   Error return
```

2.7.17 GR8_SORT

`GR8_SORT(X,XSORT,INDEX,N)`

Reorder a real*8 array using the sort indexes computed by `GR8_TRI`. Note that `X` and `XSORT` must be different (i.e. sorting cannot take place within the same array).

```
Arguments :
        X       R*8(*)  Unsorted/Sorted array           Input/Output
        XSORT   R*8(*)  Sorted array                    Work space
        INDEX   I(*)    Integer array of sorted indexes Input
                        (obtained by GR8_TRI)
        N       I       Length of arrays                Input
```

2.7.18 GR_CLIP

`LOGICAL FUNCTION GR_CLIP(clip)`

Turn on `(CLIP = .TRUE.)` or off `(CLIP = .FALSE.)` clipping of lines, and return current status in `GR_CLIP`. By default, clipping is on. Caution : some GREG subroutines force clipping off and reset it on upon exit.

2.8 The cursor routine

`GR_CURS (XU,YU,X4,Y4,CODE)`

Calls the interactive graphic cursor and returns its position when you hit any alphanumeric key on the keyboard.

```
Arguments :
        XU      REAL*8      X User coordinates    Output
        YU      REAL*8      Y User coordinates    Output
        X4      REAL*4      X Plot coordinates    Output
```

```
Y4      REAL*4       Y Plot coordinates    Output
CODE    CHARACTER*1  Character struck      Output
```

The subroutine returns `CODE = 'E'` when an error occurs.

For X-Window terminals, button 1 returns ^ , 2 returns & and 3 returns ∗.

2.9 GREG High-Level Subroutines

These routines essentially format the command line to pass it later to the `GR_EXEC` subroutine, and are thus the less efficient routines of the GREG library. They are provided essentially for user convenience because the formatting needed to use `GR_EXEC` might be tedious. Note all GREG commands have the high level equivalent, but all can be used with `GR_EXEC`.

Each subroutine corresponds to a command, and each entry corresponds to an option of that command. The subroutine and entry names are built from the 4 first characters of the corresponding command and option names. The entries must be called before the subroutine, since it is this one which effectively transmit the command to `GR_EXEC`.

The conventional type of arguments are

```
NAME            Character
ARG,ARG1,...    Real*8
IARG,IARG1,...  Integer*4
NARG            Integer*4
```

When `NARG` is present, all arguments need not be passed : trailing arguments may be omitted. `NARG` is used to give the actual number of arguments passed after `NARG` itself and it must be set precisely. All arguments before `NARG`, usually the character string `NAME`, and the argument `NARG` must always be present (even if `NAME=' '`). Note that if `ARG1` is missing, `ARG2` cannot be present. Note also that the double quotes should not be passed for character strings. For instance, the interactive command `LABEL ``A little toy''` should be replaced by `CALL GR_LABE('A little toy')`.

Example : `CALL GR_DRAW ('MARKER', 2, 0.3D0, 0.8D0)`

Note that the `ARG1` and `ARG2` arguments are here expressed as constants, and the D exponent is required as they must be Real*8 values.

The high level routines are not yet fully guaranteed as up to date. In the following description, each high level subroutine is preceeded by the assumed syntax for the equivalent command. If this syntax differs from the current internal help description, using the discrepant parts of the high level routine will cause a fatal error, but common parts of the syntax can be used safely.

For the GTVL language, the following routines are available

```
CLEAR [Argument]
        SUBROUTINE GR_CLEA(NAME)

DEVICE [Type [Descriptor]] [/OUTPUT Logical_Name]
        SUBROUTINE GR_DEVI(NAME)
        All arguments of command DEVICE being character strings, it is
        easy to concatenate them into NAME.
```

For the GREG1 language, the routines are

```
AXIS Name [A1 A2] [/LOCATION X Y] [/TICK Orientation [BIG SMALL]]
        [/LABEL Position] [/[NO]LOG] [/ABSOLUTE]
        SUBROUTINE GR_AXIS(NAME,NARG,ARG1,ARG2)         ! 28-Sep-1986
        ENTRY GR_AXIS_LOCA(NARG,ARG1,ARG2)
        ENTRY GR_AXIS_TICK(NAME,NARG,ARG1,ARG2)
        ENTRY GR_AXIS_LABE(NAME)
        ENTRY GR_AXIS_LOG
        ENTRY GR_AXIS_NOLO
        ENTRY GR_AXIS_ABSO

BOX [Arguments]
        SUBROUTINE GR_BOX(NAME)                         ! 28-Sep-1986
        ENTRY GR_BOX_ABSO

COLUMN [X Nx] [Y Ny] [E Ne] [/FILE File] [/LINES Lmin [Lmax]]
        [/TABLE TableName]
        SUBROUTINE GR_COLU(NAME)                        ! 28-Sep-1986
        As this routine is normally of no use in the library version,
        it requires you to code explicitly the remaining part of the
        command line. Hence, it should be equivalent to use
        either GR_EXEC('GREG1\COLUMN '//NAME)
        or GR_EXEC1('COLUMN '//NAME)

CONNECT [/BLANKING Bval Eval]                           ! 28-Sep-1986
        SUBROUTINE GR_CONN
        ENTRY GR_CONN_BLAN(NARG,ARG1,ARG2)

DRAW [Action [X Y]] [/USER] [/BOX N] [/CHARACTER N] [/CLIP]
        SUBROUTINE GR_DRAW(NAME,NARG,ARG1,ARG2)
        ENTRY GR_DRAW_CHAR(NARG,IARG)
        ENTRY GR_DRAW_USER
        ENTRY GR_DRAW_BOX(NARG,IARG)
        ENTRY GR_DRAW_CLIP

        Note that no support is given for DRAW TEXT and DRAW FILL_AREA,
        unless you specify in NAME all the arguments and set NARG to 0.
        DRAW TEXT can easily be replaced by GR_DRAW('RELOCATE'...)
        followed by a call to GR_LABE.

ERRORBAR NAME
        SUBROUTINE GR_ERRO(NAME)                        ! 28-Sep-1986

HISTOGRAM [/BASE [Ybase]] [/BLANKING Bval Eval]
        SUBROUTINE GR_HIST                              ! 28-Sep-1986
        ENTRY GR_HIST_BASE(ARG)
        ENTRY GR_HIST_BLAN(NARG,ARG1,ARG2)
```

```
LABEL "String" [/X] [/Y] [CENTERING N] [/APPEND]
        SUBROUTINE GR_LABE(NAME)                          ! 28-Sep-1986
        ENTRY GR_LABE_X
        ENTRY GR_LABE_Y
        ENTRY GR_LABE_CENT(IARG)
        ENTRY GR_LABE_APPE

LIMITS [Xmin Xmax Ymin Ymax] [/XLOG] [/YLOG] [/RGDATA] [/REVERSE [X] [Y]]
        [/BLANKING Bval Eval]
        SUBROUTINE GR_LIMI(NARG,ARG1,ARG2,ARG3,ARG4)    ! 28-Sep-1986
        ENTRY GR_LIMI_XLOG
        ENTRY GR_LIMI_YLOG
        ENTRY GR_LIMI_RGDA
        ENTRY GR_LIMI_REVE(NAME)
        ENTRY GR_LIMI_BLAN(NARG,ARG1,ARG2)
        Arguments of GR_LIMI can be omitted with the standard VAX
        convention to indicate that automatic limit must be computed
        for this one. For example
                CALL GR_LIMI_XLOG
                CALL GR_LIMI(4,0.d0, ,-10.d0,10.d0)
        is equivalent to
                CALL GR_EXEC1('LIMITS 0 * -10 10/XLOG')
        Presently, there is no support for the fifth argument (Angular
        unit or Absolute coordinates) and for the =, < and > possibilities
        of command LIMITS in this High Level routine. To use these
        possibilities, you should use directly GR_EXEC1('LIMITS ...').

PENCIL [N] [/COLOUR C] [/DASHED D] [/WEIGHT W]
        SUBROUTINE GR_PEN(IPEN,ICOLOUR,COLOUR,IDASH,IWEIGHT,ERROR)

POINTS [Size] [/BLANKING Bval Eval]
        SUBROUTINE GR_POIN(NARG,ARG1)
        SUBROUTINE GR_POIN_BLAN(NARG,ARG1,ARG2)

RULE [X] [Y] [/MAJOR [/MINOR]
        SUBROUTINE GR_RULE(NAME)                          ! 28-Sep-1986
        ENTRY GR_RULE_MAJO
        ENTRY GR_RULE_MINO

SET Something [Value1 [Value2]...]]]
        SUBROUTINE GR_SET(NAME,NARG,ARG1,ARG2,ARG3,ARG4) ! 28-Sep-1986

SHOW
        SUBROUTINE GR_SHOW(NAME)                          ! 28-Sep-1986

TICKSPACE SmallX BigX SmallY BigY                         ! 28-Sep-1986
```

```
        SUBROUTINE GR_TICK(NARG,ARG1,ARG2,ARG3,ARG4)
```

The following commands correspond to the GREG2\ Language.

```
EXTREMA [/BLANKING Bval Eval] [/PLOT]
        SUBROUTINE GR_EXTR                              ! 28-Sep-1986
        ENTRY GR_EXTR_BLAN(NARG,ARG1,ARG2)
        ENTRY GR_EXTR_PLOT

LEVELS List
        SUBROUTINE GR_LEVE(NAME)
        Coding the list in the required format may be funny.
        Use GR4_LEVELS or GR8_LEVELS instead, they are more convenient.

RGDATA File_Name [/SUBSET IX1 IY1 IX2 IY2]
        SUBROUTINE GR_RGDA(NAME)                        ! 28-Sep-1986
        ENTRY GR_RGDA_SUBS(NARG,IARG1,IARG2,IARG3,IARG4)

RGMAP [/ABSOLUTE Value] [/PERCENT Value] [/KEEP] [/BLANKING Bval Eval]
        [/GREY Colour Ntry] [/PENS Pos Neg]
        SUBROUTINE GR_RGMA                              ! 28-Sep-1986
        ENTRY GR_RGMA_ABSO(ARG)
        ENTRY GR_RGMA_PERC(ARG)
        ENTRY GR_RGMA_BLAN(NARG,ARG1,ARG2)
        ENTRY GR_RGMA_KEEP
        ENTRY GR_RGMA_PENS(NARG,IARG1,IARG2)
        No support is yet given for option /GREY because it is still
        rather experimental.
```

Remember that if anything available in interactive seems to be missing in the previous list, you can always use GR_EXEC to access it.

Chapter 3

Task Programming Manual

CAUTION: This section has NOT yet been FULLY updated for the Gildas Data Format Version 2 (GDFV2). Please refer to the *sic-gdfv2* document. More information is given in each section.

3.1 General Outline and Data Structure of Images

Section basically up to date

The GILDAS software is designed to work in an heterogeneous network, where each computer may have its own floating point and integer number representation and its own operating system. Images created on any computer should be accessible transparently from any other computer in the network.

To allow such a portability and yet preserve efficiency when working on a single type of machines, all files are written in the binary representation of the machine on which they were created. A library of subroutines is used to access the images and perform the necessary format conversion between the different representations of real and integer numbers. So far, 3 representations are recognized: VAX format (a dead dinosaur...), IEEE format (the so-called big-endians, like DEC DS Workstation, Intel & AMD PCs and most affordable current computers) and EEEI format (big-endians, like Sun SparcStation, IBM RS-6000 series, and other obsolescent machines...).

The images have the following organization :

- Images are basically direct access files with a logical blocksize of 512 bytes. This correspond to the system block size on VAX-VMS machines.
- The first block is a header block, defining the size of the image and all major parameters, such as World Coordinate System definition.
- The following blocks containing the image array itself.
- There may be trailing blocks for additional information. These trailing blocks are not compulsory, and may be ignored by the processing software.
- the number of logical blocks is rounded upwards to a multiple of 16 blocks, to allow reading with 8192 byte blocks.

The header starts with 'GILDAS', followed by one character which indicates the type of integer and floating point number representation

- The inferior sign ">" for swapped IEEE machines (big-endians, IBM RS-6000, SUN Sparc, etc...), with a 64-bit header (so-called GDFV2 data version)
- The superior sign "<" for non-swapped IEEE machines (little-endians) with a 64-bit header (so-called GDFV2 data version)
- Underscore "_" for VAX machines
- Dot "." for swapped IEEE machines (IBM RS-6000, SUN Sparc, etc...)
- Minus "-" for non-swapped IEEE machines (DEC DS series)

The last 3 cases are inherited from the GDFV1 data version, in which all size informations were limited to 32-bit numbers. The latest GDFV2 data version has some integers using 64 bit values to handle large data sets.

After the version/hardware sign, an extra text indicates the type of information in the image:

- `IMAGE` for all images (up to 4-D arrays)
- `UVFIL` for UV tables

This character string, being human readable, is intended for quick checks, but a more specific information about the actual type of data file is available through an integer code in the data header.

3.2 Fortran-90 access to images

Section basically up to date

Fortran-90 allows definition of data structures (derived types) which are convenient to handle data structures. Data access to GILDAS files is done by reading all or part of the data into memory, using standard FORTRAN direct I/O. The header is read into a specific Fortran derived type, the `gildas` derived part, which has a sub-type mimicking the file header plus ancillary information re-organized for convenience. The data can be read into standard Fortran arrays. Fortran pointers are provided as placeholders for this purpose in the `gildas` data type for rank 1 to 4 real arrays.

The portable version of the GILDAS software makes the distinction between an image and the "incarnation" of an image subset in memory. Each image has an associated "Image Slot" (IS), while each memory part has an associated "Memory Slot" (MS). Several memory slots may corresponds to several "windows" into a single image slot.

The Fortran-90 module `image_def` defines the `gildas` derived type.

```
!
module image_def
  use gildas_def
  use gbl_format
  use gio_headers
  !
  !
  type :: gildas
    sequence
```

```
    character(len=256)   :: 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
  !
end module image_def
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

The type `gildas_v2` closely mimicks the actual layout of the data file, but is **** not **** identical. Some of its content are provided as convenience to handle the quantities, but are **** not **** intended to be used directly in the application programs. The `ijxyz(3)` (where `xyz` can be `typ, lin, sou, sys, uni, cod`) pseudo-integer arrays in particular should never be used and may disappear in future releases (or become private). They correspond to the character strings held in the derived type `strings`.

```
  type :: gildas_header_v2
     !
     ! Spread on two blocks
     !
     ! Block 1: Basic Header and Dimension information
     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 numbe
     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 l
    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 secti
    integer(kind=4) :: mdim    = 4        !or > ! 19  Maximum number of dimensions in this dat
    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 file:     integer(kind=8) :: mini = 0           ! Rank 1 pixel of minimum
! in file:     integer(kind=8) :: maxi = 0           ! Rank 1 pixel of maximum
    !
    ! 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 lengtl
    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 sect
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: 14 used
integer(kind=4) :: void_start  !! = s_uvda + l_uvda + 2
integer(kind=4) :: version_uv = code_version_uvt_current  ! 1 version number. Will allow u
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 channel
integer(kind=4) :: lcol              ! 10 Column of last  channel
! 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 (at lest 7)
! 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 ! Back pointer to the 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)         ! Start column for each ...
! integer(kind=4) :: column_types(nlead+ntrail) /0,1,2/ ! Number of columns for each: 1 re
```

```
    ! 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)           ! 15-18 Atom description
    !
    real(kind=8), pointer :: freqs(:) => null()     ! (nchan*nstokes) = 0d0
    integer(kind=4), pointer :: stokes(:) => null() ! (nchan*nstokes) or (nstokes) = code_sto
    !
    ! back pointers to the ref,val,inc naming convention
    real(kind=8), pointer :: ref(:) => null()
    real(kind=8), pointer :: val(:) => null()
    real(kind=8), pointer :: inc(:) => null()
  end type gildas_header_v2
```

Access to images is very simple. It requires only 3 steps: *i)* to read the header from an existing file, or to create a new header, *ii)* to allocate the data, *iii)* to read or write the data. An example is given below.

```
program image_example
  use image_def                                      ! 1
  use gkernel_interfaces
  logical error
  integer ier
  character*32 name1,name2
  !
  type (gildas) :: input_image, output_image         ! 2
  real, allocatable :: dinput(:,:), doutput(:,:,:)   ! 3
  !
  call gildas_open
  call gildas_char('INPUT$',name1)
  call gildas_char('OUTPUT$',name2)
  call gildas_close
  !
  call gildas_null(input_image, type='IMAGE')        ! 4
  call sic_parsef (name1,input_image%file,' ','.gdf') ! 5
  call gdf_read_header (input_image,error)           ! 6
  if (error) then
    call gagout('E-IMAGE_EXAMPLE, Error opening input file')
    call sysexi(fatale)
  endif
  allocate(dinput(input_image%gil%dim(1),input_image%gil%dim(2), &
       stat=ier)                                     ! 7
  if (ier.ne.0) then
    call gagout('e-image_example, error allocating memory')
    call sysexi(fatale)
  endif
  call gdf_read_data (input_image, dinput, error)    ! 8
```

```
if (error) then
  call gagout('E-IMAGE_EXAMPLE, error reading input file')
  call sysexi(fatale)
endif
!
! Create an output image
!-----------------------
call gdf_copy_header (input_image, output_image)    ! 9
call sic_parsef (name2,output_image%file,' ','.gdf')! 10
output_image%gil%ndim = 3                           ! 11
output_image%gil%dim(1) = input_image%gil%dim(1)    ! 11
output_image%gil%dim(2) = input_image%gil%dim(2)    ! 11
output_image%gil%dim(3) = 4                         ! 11
allocate(doutput(output_image%gil%dim(1), &
  output_image%gil%dim(2),output_image%gil%dim(3), &
  stat=ier)                                         ! 12
if (ier.ne.0) then
  call gagout('E-IMAGE_EXAMPLE, Error allocating memory')
  call sysexi(fatale)
endif
!
! Do something with the data
doutput(:,:,3) = dinput
!
! Write the output image
call gdf_write_image(output_image,doutput,error)    ! 13
if (error) then
  call gagout('E-IMAGE_EXAMPLE, Error writing output file')
  call sysexi(fatale)
endif
!
deallocate(dinput,doutput)
end
```

1. USE the module containing the GILDAS derived type definitions
2. Define the input and output image headers
3. Define the input and output data as allocatable arrays
4. Initialize the input image header. Argument `type` is optional, and default to ``IMAGE''
5. Prepare the input file name, `INPUT_IMAGE%FILE`
6. Read the header to initialize the `INPUT_IMAGE` structure.
7. Allocate the data. Note that it is assumed here to be a 2-D array.
8. Read the data, using the information provided in the header structure (in particular the file name).

9. Define the output header, here by making a copy of the input header
10. Setup the output file name
11. Change the output header parameters as needed
12. Allocate the output image data
13. Create and write the output image

The subroutines using GILDAS headers are:

- `SUBROUTINE GILDAS_NULL(HEADER)`
  Initialize a header. **No access to header components should be done before this, as some variables are dangling pointers before initialization.**
- `SUBROUTINE GDF_COMPARE_SHAPE(FIRST,SECOND,EQUAL)`
  Compare the shape of the two images defined by the two headers
- `SUBROUTINE GDF_READ_HEADER(IMAG,ERROR)`
  Read the image header. `IMAG%FILE` must have been initialized before.
- `SUBROUTINE GDF_UPDATE_HEADER(IMAG,ERROR)`
  Update an image header: write the modified header to the image file.
- `SUBROUTINE GDF_COPY_HEADER(INPUT,OUTPUT)`
  Copy an input header structure to an output header structure
- `SUBROUTINE GDF_TRANSPOSE_HEADER(INPUT,OUTPUT,ORDER,ERROR)`
  Transpose a header according to given transposition code. This routine transposes the axes information, but not the data.

3.3 GIO API

Missing section, see the *sic-gdfv2* document.

3.4 Obsolete Fortran-77 access routines

Caution: These routines are obsolete. Documentation only provided for historical reasons. It may disappear at some point or be moved to another document.
In Fortran-77, neither virtual memory handling, nor data structures, are part of the language. We had written a number of routines to access images. These routines are more complex than the new Fortran-90 method of access, and are provided here just for completeness.

To read a file requires six basic steps:

- allocation of an *Image Slot* using `GDF_GEIS`.
- connection of an image to the slot using `GDF_REIS` or `GDF_WRIS`.
- reading the header using `GDF_RHSEC`
- connection of a *Memory Slot* to the image (or a subset of it) using `GDF_GEMS`

- once the *Memory Slot* is no longer needed, disconnection using `GDF_FRMS`
- once the image slot is no longer required, disconnection using `GDF_FRIS`

Creation of a new file is slightly different:

- allocation of an *Image Slot* using `GDF_GEIS`
- preparatin of the header using e.g. `GDF_WHSEC`
- creation of the image to the slot using `GDF_CRIS`. The header is written at this stage.
- connection to the image (or a subset of it) using `GDF_GEMS`
- once the *Memory Slot* is no longer needed, disconnection using `GDF_FRMS`. This is the step which actually does the writing.
- once the image slot is no longer required, disconnection using `GDF_FRIS`

3.4.1 Image Slot Handling

```
SUBROUTINE GDF_GEIS (IS,ERROR)

      GDF  GEt Image Slot
           IS      I       Slot number                    Output
           ERROR   L       Error flag                     Output
```

`GDF_GEIS` returns a unique image slot number IS, for further use in all other `GDF_xxxx` routines.

```
SUBROUTINE GDF_CLIS (IS,ERROR)

      GDF  CLose Image Slot
           IS      I       Slot number                    Input
           ERROR   L       Error flag                     Output
```

`GDF_CLIS` close an image slot IS, disconnecting all associated memory slot after flushing all pending updates on the image. The image slot IS stays reserved, but available for further use.

```
SUBROUTINE GDF_FRIS (IS,ERROR)

      GDF  FRee Image Slot
           IS      I       Slot number                    Input
           ERROR   L       Error flag                     Output
```

`GDF_CLIS` free an image slot IS, disconnecting all associated memory slot after flushing all pending updates on the image. The image slot IS should no longer be referenced in any call after this operation.

```
INTEGER FUNCTION GDF_STIS(IS)

      GDF  STatus of Image Slot
           IS      I       Slot number                    Input
```

Return status of slot: -1 No such slot, 0 Empty (Not allocated), 1 Read (Opened for ReadOnly), 2 Full (Allocated but not opened), 3 Write (Opened for ReadWrite).

3.4.2 Image Connection

Once a slot is reserved, it must be associated with an image. The following 4 routines connect an image to a specified image slot IS, in one of the 4 possible access modes (CReate, EXtend, REad only, Write and Read). No part of the image is immediately accessible after that, but the image slot is ready for memory connection using GDF_GEMS.

```
SUBROUTINE GDF_CRIS (IS,GTYPE,NAME,FORM,SIZE,ERROR)

      GDF  CReate Image Slot
      Arguments:
           IS      I       Slot number                     Input
           GTYPE   C*(*)   Image type                      Input
           NAME    C*(*)   File name                       Input
           FORM    I       Image format (Real,Integer...)  Input
           SIZE    I       Image size                      Input
           ERROR   L       Error flag                      Output
```

Creates a new file from the information available in image slot IS. The internal header should have been defined (through calls to GDF_WHSEC) before, and is written on the new file header.

```
SUBROUTINE GDF_EXIS (IS,GTYPE,NAME,FORM,SIZE,ERROR)

      GDF  EXtend Image Slot
      Arguments:
           IS      I       Slot number                     Input
           GTYPE   C*(*)   Image type                      Input
           NAME    C*(*)   File name                       Input
           FORM    I       Image format (Real,Integer...)  Input
           SIZE    I       Image size                      Input
           ERROR   L       Error flag                      Output
```

This routine is used to change the size of an image, usually by increasing the last dimension of the image. The new internal header must have been defined before in image slot IS, by use of the GDF_WHSEC routine.

```
SUBROUTINE GDF_REIS (IS,GTYPE,NAME,FORM,SIZE,ERROR)

      GDF  REad Image Slot
      Arguments:
           IS      I       Slot number                     Input
           GTYPE   C*(*)   Image type                      Output
           NAME    C*(*)   File name                       Input
           FORM    I       Image format (Real,Integer...)  Output
           SIZE    I       Image size                      Output
           ERROR   L       Error flag                      Output
```

This routine is used to open an existing image for ReadOnly operations. The image is available for file sharing.

```
SUBROUTINE GDF_WRIS (IS,GTYPE,NAME,FORM,SIZE,ERROR)

      GDF  WRite Image Slot
      Arguments:
           IS      I       Slot number                      Input
           GTYPE   C*(*)   Image type                       Output
           NAME    C*(*)   File name                        Input
           FORM    I       Image format (Real,Integer...)   Output
           SIZE    I       Image size                       Output
           ERROR   L       Error flag                       Output
```

This routine is used to open an existing image for ReadWrite operations. The image is not available for file sharing.

3.4.3 Memory Connection

After an image is connected, a part of it should be brought into virtual memory. The access type for this virtual memory area (ReadOnly or ReadWrite) depends on the type of image connection.

```
      SUBROUTINE GDF_GEMS (MS, IS, BLC, TRC, ADDR, FORM, ERROR)
-----------------------------------------------------------------------
 GDF  Get Memory Slot
      MS      I       Memory Slot number             Output
      IS      I       Image Slot number              Input
      BLC     I(4)    Bottom Left Corner             Input
      TRC     I(4)    Top Right Corner               Input
      ADDR    I       Virtual memory address         Output
      FORM    I       Type of data                   Input
      ERROR   L       Logical error flag             Output
-----------------------------------------------------------------------
```

GDF_GEMS reads a subset of the image connected to slot IS into a memory area (memory slot MS) located at address ADDR. The image subset is "incarnated" into the specified data type FORM, which may be different from the image data type. When BLC(i) and TRC(i) are set to zero, they default to the current image dimension (i.e BLC(i)=1 and TRC(i)=Dim(i)). After this routine, virtual memory address ADDR is the start of an array of type FORM containing an incarnation of the image subset that can be used for further processing.

```
      SUBROUTINE GDF_FRMS (MS,ERROR)
-----------------------------------------------------------------------
 GDF  FRee Memory Slot
-----------------------------------------------------------------------
```

GDF_FRMS Frees the memory slot MS. If the connected image was connected with write access, the image is updated. The memory slot is no longer available after this operation.

```
      SUBROUTINE GDF_UPMS (MS,ERROR)
-----------------------------------------------------------------------
 GDF  UPdate Memory Slot
-----------------------------------------------------------------------
```

`GDF_UPMS` Updates the image connected to the memory slot MS. The memory slot is left unmodified. This routine is provided for safety measures.

The I/O (or mapping) are thus only done in the `GDF_something` routines, in a system dependent way. All data format translation (incarnation in a different type, or machine dependent data types) are done at this level. This allows a transparent operation in an heterogeneous environment.

3.5 Creating GILDAS Tasks

GILDAS tasks (e.g. for task "`taskname`") are composed of 4 different parts :

- The program, `taskname.exe`.
- A help text file, `taskname.hlp`
- The initialization file, `taskname.init`, read by commands `RUN` and `SUBMIT` to get from the user the task input parameters.
- Optionally, a checker file, `taskname.check`, read by commands `RUN` and `SUBMIT`, to check the input parameters and pass them to the task. It is optional: SIC will use a suitable default derived from the initialisation file when the checker file does not exist.

Tasks and their associated files are searched for in the `task#dir:` search path (by default ``./;gag_tasks:''`).

The structure of the program should always be the same :

- An input parameters definition part. No image creation or access should be made before all parameters are defined.
- An input and output image mapping part. If possible all necessary images should be defined in this part.
- One or more subroutines performing the processing. No new image should be created or accessed in the subroutines. The subroutines should preferably not make any reference to any common.
- A cleanup part. An error status should be returned to the system in case of error through a call to `SYSEXI`.

3.6 A Template Task

The following program is a typical task with two input images and an output one. The easiest way to create a new task is just to start with this working template and modify what is needed.

Note that this sample task uses integer of kind=4, i.e. 32-bit numbers, to handle some dimensions, so it will be limited to < 2 Gbyte data size. Changing to kind=8 (in main program and subroutines...) would remove this limitation.

3.6.1 Source code

```
program combine
  !-------------------------------------------------------------------------
  ! GILDAS Combine in different ways two input images
  !        (or data cubes)...
  !-------------------------------------------------------------------------
  use image_def
  use gkernel_interfaces
  use gbl_format
  !
  character(len=filename_length) :: 80 namex,namey,namez
  character(len=20) :: code
  logical error
  real ay,az,ty,tz,b,c
  type (gildas) :: hx, hy, hz
  real, allocatable :: dx(:,:), dy(:,:), dz(:)
  integer(kind=4) :: i, j, n, m
  integer :: ier
  !
  call gildas_open
  call gildas_char('Z_NAME$',namez)
  call gildas_real('Z_FACTOR$',az,1)
  call gildas_real('Z_MIN$',tz,1)
  call gildas_char('Y_NAME$',namey)
  call gildas_real('Y_FACTOR$',ay,1)
  call gildas_real('Y_MIN$',ty,1)
  call gildas_char('X_NAME$',namex)
  call gildas_real('BLANKING$',b,1)
  call gildas_real('OFFSET$',c,1)
  call gildas_char('FUNCTION$',code)
  call gildas_close
  !
  n = len_trim(namez)
  if (n.eq.0) goto 100
  call gildas_null(hz)
  call sic_parsef(namez(1:n),hz%file,' ','.gdf')
  call gdf_read_header(hz,error)
  if (error) then
    call gagout('F-COMBINE,  Cannot read input file')
    goto 100
  endif
  !
  n = len_trim(namey)
  if (n.eq.0) goto 100
  call gildas_null(hy)
  call sic_parsef(namey(1:n),hy%file,' ','.gdf')
  call gdf_read_header(hy,error)
```

```
if (error) then
  call gagout('F-COMBINE,  Cannot read input file')
  goto 100
endif
!
if (hz%gil%eval.ge.0.0) hz%gil%eval =   &
  max(hz%gil%eval,abs(hz%gil%bval*1e-7))
if (hy%gil%eval.ge.0.0) hy%gil%eval =   &
  max(hy%gil%eval,abs(hy%gil%bval*1e-7))
!
! Check input dimensions
do i=1,gdf_maxdims
  if (hy%gil%dim(i).ne.hz%gil%dim(i)) then
    n = 1
    do j=i,gdf_maxdims
      n = n*hz%gil%dim(j)
    enddo
    if (n.ne.1) then
      call gagout('F-COMBINE,  Input images are non coincident')
      goto 100
    else
      call gagout('W-COMBINE,  Combining a cube with a plane')
    endif
  endif
enddo
!
call gdf_copy_header(hy,hx)
n = len_trim(namex)
if (n.eq.0) goto 100
call sic_parsef(namex(1:n),hx%file,' ','.gdf')
hx%gil%blan_words = 2
hx%gil%bval = b
hx%gil%eval = 0.0
hx%gil%extr_words = 0  ! No extrema computed
!
! Allocate the arrays. Note that the allocated arrays do not conform
! to the shape of the images: DZ is allocated as a 1-D array, DX,DY
! as 2-D arrays, possibly of second dimension 1.
!
n = hz%loca%size
m = hx%loca%size/hz%loca%size
allocate(dx(n,m),dy(n,m),dz(n),stat=ier)
if (ier.ne.0) then
  call gagout('F-COMBINE,  Input images are non coincident')
  goto 100
endif
!
```

```
  ! Read the input data
  call gdf_read_data(hz,dz,error)
  call gdf_read_data(hy,dy,error)
  !
  if (code.eq.'ADD') then
    call add002(dz,dy,dx,   &
      n,m,   &
      hz%gil%bval,hz%gil%eval,az,tz,   &
      hy%gil%bval,hy%gil%eval,ay,ty,   &
      hx%gil%bval,c)
  elseif (code.eq.'DIVIDE') then
    call div002(dz,dy,dx,   &
      n,m,   &
      hz%gil%bval,hz%gil%eval,az,tz,   &
      hy%gil%bval,hy%gil%eval,ay,ty,   &
      hx%gil%bval,c)
  elseif (code.eq.'MULTIPLY') then
    call mul002(dz,dy,dx,   &
      n,m,   &
      hz%gil%bval,hz%gil%eval,az,tz,   &
      hy%gil%bval,hy%gil%eval,ay,ty,   &
      hx%gil%bval,c)
  elseif (code.eq.'OPTICAL_DEPTH') then
    call opt002(dz,dy,dx,   &
      n,m,   &
      hz%gil%bval,hz%gil%eval,az,tz,   &
      hy%gil%bval,hy%gil%eval,ay,ty,   &
      hx%gil%bval,c)
  else
    call gagout('Invalid operation code '//code)
    goto 100
  endif
  !
  ! Write ouput file
  call gdf_write_image(hx,dx,error)
  !
  stop 'S-COMBINE,  Successful completion'
  !
  100     call sysexi (fatale)
end
!
subroutine add002(z,y,x,n,m,bz,ez,az,tz,by,ey,ay,ty,bx,c)
  !---------------------------------------------------------------------
  ! GDF Internal routine
  !     Linear combination of input arrays
  !     X = Ay*Y + Az*Z + C
  ! Arguments
```

```
!     Z               R*4(*)  Input array (N)
!     Y               R*4(*)  Input array (N,M)
!     X               R*4(*)  Output array (N,M)
!     N,M             I       Dimensions of arrays
!     BX,BY,BZ        R*4     Blanking values
!     EY,EZ           R*4     Tolerance on blanking
!     AZ,AY           R*4     Multiplicative factor of array Z, Y
!     TZ,TY           R*4     Threshold on Z,Y
!     C               R*4     Additive constant
!-----------------------------------------------------------------------
integer :: n                        !
real :: z(n)                        !
integer :: m                        !
real :: y(n,m)                      !
real :: x(n,m)                      !
real :: bz                          !
real :: ez                          !
real :: az                          !
real :: tz                          !
real :: by                          !
real :: ey                          !
real :: ay                          !
real :: ty                          !
real :: bx                          !
real :: c                           !
! Local
integer :: i,k
!
do k=1,m
  do i=1,n
    if (abs(z(i)-bz).gt.ez .and. abs(y(i,k)-by).gt.ey   &
   &        .and. z(i).gt.tz .and. y(i,k).gt.ty) then
      x(i,k) = ay*y(i,k) + az*z(i)  +  c
    else
      x(i,k) = bx
    endif
  enddo
enddo
end
```

3.6.2 Initialization file

The initialization file is a standard SIC procedure containing only commands from the languages `TASK\` or `SIC\`. `TASK\` language must be specified explicitly. The command syntax is always the following

`TASK\Type_of_parameter "Prompt string" Parameter$[Dimension]`

where

- `Type_of_Parameter` can be `CHARACTER, REAL, INTEGER or LOGICAL`

- `"Prompt string"` is a text used as a prompt if required
- `Parameter$` is the parameter name, the parameter being a standard SIC variable, possibly with one dimension. It is recommended to include a `$` as last character to avoid possible confusion with user declared variables.

```
!
! Combine.INIT
TASK\FILE "First input map" Z_NAME$
TASK\REAL "Scaling factor" Z_FACTOR$
TASK\REAL "Threshold" Z_MIN$
TASK\FILE "Second input map" Y_NAME$
TASK\REAL "Scaling factor" Y_FACTOR$
TASK\REAL "Threshold" Y_MIN$
TASK\FILE "Output map" X_NAME$
TASK\REAL "New blanking value" BLANKING$
TASK\REAL "Output offset" OFFSET$
TASK\CHARACTER "Function" FUNCTION$
TASK\GO                     ! Must be last command
```

The parameter names, types and dimensions must correspond to those declared in the source code. All parameters must be defined. Ordering may be important (see Checker File).

3.6.3 The HELP file

A standard help file should be prepared for each task. The format follows that of SIC help files. Topics are identified by a "1" in first column, and subtopics by a "2". The help file must start with the task name as main topic, and must have a subtopic for each parameter. More subtopics may exist. The help file must be in the same place as the executable image, and have file type `.hlp`. For the example above, `combine.hlp` contains

```
1 COMBINE
                                COMBINE

     It makes "combinations" of two input images to  produce  a  third
one.   The two input images may have the same dimensions, or the first
one (Z one) may have less dimensions than the second (Y) one.  In  the
latter case, combinations will occur for all the extra planes of the Y
image.  For example you can divide all the plane of an input  (Y)  3-D
cube  by  a 2-D (Z) image, provided each plane of the cube matches the
single image...

     Operations are
        ADD             X = Ay*Y + Az*Z + C
        MULTIPLY        X = Ay*Y * Az*Z + C
        DIVIDE          X = Ay*Y / Az*Z + C
provided Y > Ty and Z > Tz, where Ty and Tz and thresholds
set by parameters YMIN$ and ZMIN$.
```

```
2  Z_NAME$
This is the  name  of  the  input  map  with  the  smaller  number  of
dimensions.
2  Z_FACTOR$
This is a scaling factor for map Z_NAME$.
2  Z_MIN$
This is a threshold on map Z_NAME$.
2  Y_NAME$
This is  the  name  of  the  input  map  with  the  larger  number  of
dimensions.
2  Y_FACTOR$
This is a scaling factor for map Y_NAME$.
2  Y_MIN$
This is a threshold on map Y_NAME$.
2  X_NAME$
This is the name of the output map.
2  BLANKING$
This is the blanking value chosen for the output map.
2  OFFSET$
This is an offset added to the output map.
2  FUNCTION$
Selected operation.  Possible operations are ADD, MULTIPLY, DIVIDE  (Y
by Z).
```

3.6.4 Checker File

The checker file is optional. If it does not exist, SIC will create one writing all the task input variables in the order in which they have been defined.

The checker file is another SIC procedure containing only `TASK\` and `SIC\` commands, which tests the validity of the input parameters (to avoid submission of tasks with bad parameters), and writes the parameters. Checking is optional and can be done using SIC facilities. All parameters are known SIC variables. Writing is done using command `TASK\WRITE`, and the ordering must match the source code. The task is initiated by the `TASK\GO` command.

```
!
! Combine.CHECK
SIC\IF (FUNCTION$.EQ."ADD") THEN
    SIC\SAY "Computing X_NAME = -
    'Y_FACTOR$'*Y_NAME + 'Z_FACTOR$'*Z_NAME + 'OFFSET$'"
SIC\ELSE IF (FUNCTION$.EQ."DIVIDE") THEN
    SIC\SAY "Computing X_NAME = -
    'Y_FACTOR$'*Y_NAME / 'Z_FACTOR$'*Z_NAME + 'OFFSET$'"
SIC\ELSE IF (FUNCTION$.EQ."MULTIPLY") THEN
    SIC\SAY "Computing X_NAME = -
    'Y_FACTOR$'*Y_NAME * 'Z_FACTOR$'*Z_NAME + 'OFFSET$'"
SIC\ELSE
    SIC\SAY "Invalid operation 'FUNCTION$'"
    SIC\RETURN   ! Return without a GO command : no submission
```

```
SIC\ENDIF
!
TASK\WRITE Z_NAME$
TASK\WRITE Z_FACTOR$
TASK\WRITE Z_MIN$
TASK\WRITE Y_NAME$
TASK\WRITE Y_FACTOR$
TASK\WRITE Y_MIN$
TASK\WRITE X_NAME$
TASK\WRITE BLANKING$
TASK\WRITE OFFSET$
TASK\WRITE FUNCTION$
TASK\GO                     ! Effectively RUNs or SUBMITs the task.
```

3.7 Debugging Tasks

In addition to the predefined directory, `GILDAS_RUN:`, another directory `GILDAS_LOCAL:` is also searched for tasks by commands `VECTOR\RUN` and `VECTOR\SUBMIT`. This area is searched before `GILDAS_RUN:`. If the required task is found here, the initialisation, checker and help files should also be present in the same directory. This feature allows to have experimental or user-private tasks.

Although the tasks are supposed to be non-interactive programs spawned or submitted from a main program, interactive use is possible for debugging purpose. If activated interactively, a task will ask one question for each parameter, specifying the parameter type, the parameter name, and the parameter dimension. No more information will be available. The initialization and checker file are not needed for this.

Moreover since all algorithms can be standard subroutines due to the use of virtual memory, the GILDAS interface can be done easily after debugging the algorithm.

Chapter 4

GTV Programming Manual

4.1 Concept

The GTVIRT is a fast low-level Graphic library, allowing to develop Graphic programs in a completely device independent way. The GTVIRT library involves the following concepts:

- *the Plot Page*
  The Plot Page is a virtual workspace on which all graphic items will be (virtually) drawn. The Plot Page units are "virtual centimeters". Actual drawing on a graphic device is usually done with an automatic scaling factor to match the Plot Page to the device. For hardcopy outputs, an exact matching between the Plot Page unit and centimeters is possible.

- *Segments*
  Graphic segments are the smallest separable entities in a drawing. Segments are named, and can be edited to change their aspects. Although segments are usually associated with a single, easily identified, part of the drawing (e.g. a curve, or a label, or a bitmap image), they can contain anything. The segmentation is not defined by the GTVIRT, but left to the calling program. Segments are named uniquely to help identifying them.

- *Coordinate Systems* Two coordinate systems are available within the GTVIRT: a Page coordinate system (whose units are centimeters), and a User coordinate system. Drawing can be done in any of these systems.

- *Directories*
  Directories are special graphic segments used to group in a logical way ensembles of segments. A complete directory tree can be specified in a drawing. As for the segments, the structure of the drawing is defined by the calling program, not by the GTVIRT library.

  Each directory has its own User coordinate system. This feature can be used to map several coordinate systems to different or similar regions of the Plot Page.

- *Devices*
  The GTVIRT is (completely ?) device independent. A drawing can be prepared and visualized separately. The GTVIRT allows display on a large variety of graphic devices.

- *Metafiles*
  Directories and directory trees of a Plot can be saved on a (binary) metafile, complete with all graphic segments and user coordinate systems. Metafiles can be imported as Subdirectories in any Plot.

- *Hardcopy*
  "Hard"copy on paper-like Graphic devices such as pen plotters, laser printers, etc..., are completely transparent in the GTVIRT. "Soft" hardcopy on files in industry standard graphic languages such as PostScript or HPGL is possible too.

- *Windows*
  Windows are available on some devices. This includes X-Window terminals, and MS-Windows screen. *Windows* can be attached to any directory. Several windows can be attached to the same directory. A *Window* displays all the sub-directories depending on the directory to which it is attached.

- *Plotting Depth*
  Each segment can be understood as an opaque or transparent plot. The order in which opaque segments are plotted can be controlled by their plotting depth. This feature is important when plotting bitmaps or drawing colour filled curves.

- *8-bit images*
  are supported on Windows graphic displays (X-Window or MS-Window or MAC).

The GTVIRT normally operates in a buffered mode. All drawings command a written into an internal metacode, and only transmitted to the plotting device when required by the calling program. In addition, for applications which require an unbuffered plot, an *immediate* mode is available.

4.2 Programming

- Initialisation
  The GTVIRT library is initialized by a a call to
  `SUBROUTINE INIT_GTVIRT`
  followed by a call to `SUBROUTINE GTINIT(LX,LY,LUNG,LUNH,NAME,USER_ROUTINE`
  where

  - `LX,LY` are the page dimensions
  - `LUNG,LUNH` are logical unit numbers for graphic output and hardcopies, to be supplied by the caller
  - `NAME` is a character string giving the name of the top directory of the drawings. `NAME` should start with a < sign.
  - `USER_ROUTINE` is a user-supplied subroutine called when moving from a sub-directory to the other. It must provide to the GTVIRT information about coordinate systems. The only suitable routine is available in the GREG program, and is named `GREG_USER`

- The `GTVIEW` subroutine

  The `GTVIEW` subroutine is one of the main control routine in the library. It is used to send drawing commands from the internal metacode to the graphic device, but also to performs other actions on the metacode.

  `SUBROUTINE GTVIRT(Mode)`

  where `Action` is a character string which can be

- 'Append'
  Plot all metacode starting from last drawn vector
- 'Rewind'
  Clear the screen, and plot the whole metacode from first vector
- 'Update'
  Update the screen (all windows) if needed
- 'Limits'
  Recompute the plot limits (Bounding Box)
- 'Purge'
  Delete images associated to all windows
- 'Delete'
  Delete images associated to the current window
- 'Zap'
  Delete cache-bitmap associated to images
- 'Sleep'
  Set screen update off
- 'Wake'
  Set screen update on

4.3 Basic Sequence

GREG and the GTVIRT are intimately related, and it is not recommended to use the GTVIRT without GREGȦccordingly, the programming example given below also uses some (primitive) GREG subroutines. The basic drawing sequence is

```
CALL GR_SEGM (Segment_Name,Error)   ! 1
CALL GTPOLYL (N,Xarray,Yarray)      ! 2
CALL ...                            ! 3
CALL GTVIEW('Append')               ! 4
```

1. Open a new graphic segment. This routines is an "intelligent" routine checking wether pen attributes have been changed, and calling both `GTSEGM` (the basic segment creation routine) and `GTEDIT` (which defines the segment attributes like pen colour, dashed pattern, thickness).
2. Use any drawing routine you wish, e.g. an polyline. The drawing commands go to the internal metacode only at that time.
3. ...
4. Update the graphic screen with all the new drawing commands which have been put in the metacode since last call to `GTVIEW('Append')`

Further buffering between graphic segments can be obtained by enclosing a set of complete sequences like the above one between a call to `GTVIEW('Sleep')` and `GTVIEW('Wake_up')`. Drawing to the screen will then only happen when the call to `GTVIEW('Wake_up')` is made. For compatibility with other subroutines which may also perform their own `Sleep / Wake_up` control, it is recommended to use the logical function `GTSTAT` instead, e.g.

```
SLEEP = GTSTAT ('Sleep')

  ... calls to GTVIRT ...

IF (.NOT.SLEEP) CALL GTVIEW('Wake_up')
```

4.4 Plot Structuration and multi-window applications

For some complex applications, it is useful to be able to create several graphic windows and display different plots in each of them. Such a control over the plot structure is preferably done using the GTVL command language, by appropriate calls to subroutine GR_EXECL.

SUBROUTINE GR_EXECL(Command)
execute a command from the GTVL command language.

The most relevant commands to structure a plot are

- CREATE DIRECTORY <NAME to create new top directory called <NAME
- CHANGE DIRECTORY <NAME to move to this (top) directory
- CREATE WINDOW to create a new window associated to the current directory
- CHANGE POSITION Code to move the current window a given position in the screen.
- CLEAR WINDOW to clear a window
- CLEAR TREE to erase the tree linked to the current directory (from the current top directory).
- CREATE DIRECTORY NAME to create new sub directory
- CHANGE DIRECTORY NAME to move to this sub directory

Refer to the GREG manuals for detailed help on these commands.

In addition, subroutine GTEXIST is useful to check whether a given directory already exists.

4.5 Subroutines

- gtchar
  Draws a character string.
- gtclal
  Clear alphanumeric screen. On Windows applications, transfer focus to the (current) graphic window, and raise in on top of others to display it.
- gtclear
  Erase the whole plot, all directory structures, destroy all associated windows, etc... An empty top directory is then re-created
- gtclos
  Close the current graphic device
- gtclpl
  Raise alphanumeric window, and returns focus to it.
- gtcurs
  Call the cursor

- gtdls
  Delete the last graphic segment
- gtdraw
  Draw a vector from current pen position to the specified point.
- gtedit
  Edit the current segment properties.
- gterflag, gterrgto, gterrtst
  Control error status of the library
- gtexist
  Controls the existence of a named sub-directory.
- gthard
  Create a hardcopy
- gtinit
  Initialize the GTVIRT drawing space and the GTVL language
- gtopen
  Open a graphic device
- gtpolyl
  Draw a set of lines.
- gtreloc
  Move current pen position to specified coordinates.
- gtsegm
  Open a new segment
- gtview
  Activate the drawing
- gtwhere
  Returns current pen position
- gtg_charlen
  Computes character string length
- gtg_charsiz
  Returns character size
- gtg_curs
  Returns cursor existence
- gtg_screen
  Returns clipping parameters
- gtg_open
  Returns device parameters

- gtstat
  Change GTVIRT mode (Sleep or Wake_up), and returns previous mode.
- gtv_newimage
  Create a new image slot
- gtv_image
  Draw a bitmap to an image slot
- gtv_numimage
  Get a free image slot
- gtv_majimage
  Define parameters of an image slot
- gtv_fillpoly
  Fill a closed polygon.
- init_gtvirt
  Initialize the GTVIRT library
- exit_clear
  Quick exit, to be used at program completion.
- run_gtvl
  Dispatch the GTVL commands to appropriate subroutines
- exec_gtvl
  Execute a GTVL language command or command procedure

Immediate routines are used to produce immediate actions. They use the immediate pen. These subroutines are

- gti_beep: beep
- gti_clear: clear the current window
- gti_draw: draw line to current
- gti_out: flush the normal drawing buffer.
- gti_pen: select the immediate pen
- gti_polyl: draw a polyline
- gti_reloc: relocate the immediate pen
- gti_where: returns immediate pen position

Index