Programming in GILDAS

a GILDAS working group software

October 22, 2001

GILDAS
Programming Guide
Version 1.0

The GILDAS working group is a collaborative project of the Observatoire de Grenoble (1,3) and IRAM (2), and comprises: F. Badia², D. Broguière², G. Buisson¹, L. Desbats¹, G. Duvert¹, T. Forveille¹, R. Gras³, S. Guilloteau¹,², R. Lucas¹,², R. Neri² and P. Valiron¹.

(1) Laboratoire d’Astrophysique
Observatoire de Grenoble
BP 53 X
414 Rue de la Piscine
F-38402 Saint Martin d’Hères CEDEX

(2) Institut de Radio Astronomie Millimétrique
300 Rue de la Piscine
F-38406 Saint Martin d’Hères

(3) CEPHAG
Observatoire de Grenoble
F-38402 Saint Martin d’Hères CEDEX

Contributions from and invaluable discussions with J. Cernicharo, P. Begou³, S. Delahaye³, A. Dutrey¹,², C. Rahane¹, P. Monger, J.L. Monin¹ and all GILDAS users are gratefully acknowledged.
# Contents

**1 Introduction** .............................................. 4

**2 SIC Programming Manual** .................................. 5
  2.1 Introduction ............................................. 5
  2.2 Initializing SIC: The Command Language Structure ....... 6
  2.3 The Program Structure .................................... 7
  2.4 The Help File ............................................. 11
  2.5 Retrieving Arguments ...................................... 11
  2.6 The Library Version ....................................... 12
    2.6.1 Creating the Library Interface ....................... 12
    2.6.2 Library Only Mode .................................... 13
    2.6.3 Reserved logical unit numbers: VMS .................. 14
    2.6.4 Reserved logical unit numbers: UNIX ................. 14
    2.6.5 Linking on VMS ....................................... 14
    2.6.6 Linking on UNIX ...................................... 15
  2.7 Using Variables ........................................... 15
    2.7.1 Definition ............................................ 15
    2.7.2 Assignment and Examination ......................... 16
    2.7.3 Mathematical Formula Handling ...................... 17
    2.7.4 Deleting Variables ................................... 17
  2.8 Using Functions ........................................... 18
  2.9 SIC Callable Routines .................................... 18
    2.9.1 Monitor interface routines .......................... 18
    2.9.2 SIC Arguments Retrieving Routines .................. 21
    2.9.3 Command Line Interpreter Subroutines .............. 22
    2.9.4 All Purpose General Subroutines .................... 23
    2.9.5 Symbol Manipulation Routines ........................ 23

**3 GreG Programming Manual** ................................ 24
  3.1 Interaction with FORTRAN programs ...................... 24
  3.2 Basic Routines .......................................... 25
  3.3 Linking .................................................. 26
    3.3.1 UNIX systems ......................................... 26
    3.3.2 VMS systems .......................................... 26
  3.4 Running .................................................. 27
  3.5 Example .................................................. 27
  3.6 Array Transfer ........................................... 27
    3.6.1 GR4_GIVE - GR8_GIVE ................................ 27
    3.6.2 GR4_GET - GR8_GET ................................... 27
    3.6.3 GR4_RGIVE - GR8_RGIVE ............................... 27
    3.6.4 GR4_LEVELS - GR8_LEVELS ............................ 28
  3.7 Immediate Routines ...................................... 28
    3.7.1 GR4_SEGM ............................................. 28
    3.7.2 GR4_OUT .............................................. 28
    3.7.3 GR4_DRAW - RELOCATE ................................. 28
    3.7.4 GR4_DRAW - GRELOCATE ............................... 28
    3.7.5 GR4_PHYS_USER - GR8_PHYS_USER ...................... 29
    3.7.6 GR4_USER_PHYS - GR8_USER_PHYS ..................... 29
    3.7.7 GR4_CONNECT - GR8_CONNECT .......................... 29
    3.7.8 GR4_HISTO - GR8_HISTO ............................... 29
    3.7.9 GR4_MARKER - GR8_MARKER ............................. 29
    3.7.10 GR4_CURVE - GR8_CURVE .............................. 29
1 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.
2  SIC Programming Manual

2.1 Introduction

SIC (*) is a command line interpreter, written in FORTRAN and callable as a subroutine by any program. It provides a command language rather similar to the VAX-VMS DCL 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
- keypad editing of command lines on VT200 compatible devices

NEW!: GUI interface on Motif systems

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 the logical unit 5 (standard input). This is call 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 Unix-like systems, and MS-Windows (perhaps VMS, too).

The section 2.2 describes the command vocabulary structure, and the initialization sequence of SIC. Section 2.3 gives the archetype of the structure of a program using SIC as its monitor. Section 2.4 indicates how to write the corresponding HELP files. Section 2.5 describes how to retrieve arguments from a SIC command line. Section 2.6 indicates how to complete the program so that it can also be a library of routines. Finally, Section 2.7 indicates how to use SIC variables, and Section 2.9.1 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.
2.2 Initializing SIC: The Command Language Structure

SIC is a multi-language interpreter. Each language must be initialized by means of an appropriate call to the routine SIC_LOAD. This routine has the following calling list:

```
SIC_LOAD(LANG, HELP, MCOM, VOCAB, VERSION)
```

LANG is a Character constant (less than 12 characters) 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*12 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 SIC. The following is the DATA initialization statement of SIC itself given as an example of language.

```
CHARACTER*12 VOCAB(MVOCAB)
INTEGER SIC_COMMANDS, USER_COMMANDS
PARAMETER (SIC_COMMANDS=34)
PARAMETER (USER_COMMANDS=MVOCAB-SIC_COMMANDS)
DATA VOCAB/
   'BREAK',      'CONTINUE',      '$DCL',       '/PROCESS',
   'DEFINE',      '/GLOBAL',      'DELETE',      '/SYMBOL',
   '/VARIABLE',   '#EDIT',        'ELSE',       'ENDIF',
   'EXAMINE',     '/GLOBAL',      '#EXIT',       'FOR',
   '/WHILE',      '*HELP',        'IF',         'LET',
   '/NEW',        '/PROMPT',      'NEXT',       'ON',
   '/PAUSE',      '#QUIT',        '#RECALL',    'RETURN',
   'SAY',         'SIC',          '$SYMBOL',    '/INQUIRE',
   '*TYPE',       '0',           USER_COMMANDS* '/
```

The first character code is interpreted as follows:

- **<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_LINE). 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.

Finally, VERSION is a character string which must contain the version number, and may contain the date of last modification, the name of the programmer, etc. The following example is recommended for optimum presentation (respect the alignment of the date for different languages):
In addition to setting the vocabulary, the programmer must decide the program prompt, the log file and the Stack usage. This is entirely optional because SIC provides default setups, and it is done using routine

\texttt{SIC\_OPT (PROMPT, LOGFILE, MEMORY)}

where PROMPT is a character string (less than 8 characters) defining the prompt (SIC adds automatically the caret > at end of prompt), LOGFILE is a character string defining the Log\_File for the session and MEMORY is a logical flag defining the Stack usage. There are two modes for Stack usage: the first mode defines the Stack as an internal procedure which is entirely at the disposition of the user (MEMORY=.FALSE.), the second mode defines the stack as an internal Log\_File, in which almost all commands (with the restriction in the command code) typed at the top level in Read Mode are automatically written to the Stack (MEMORY=.TRUE.). This mode provides a playback capability for interactive sessions. Modes can be commuted at run time using SIC command SIC\_MEMORY.

Default values are \texttt{SIC'}, \texttt{LOG'} and \texttt{.TRUE.} respectively. Hence, if you use the defaults, the Log\_File can also be defined by an assignment to logical name LOG. The call to SIC\_OPT is completely effective if placed before any call to SIC\_LOAD. Otherwise, only the prompt will be changed.

### 2.3 The Program Structure

In this section a complete example is described. The language name is set to USER, and it is highly recommended to use the same normalization conventions for the subroutine names. The example takes advantage of the multi-language capabilities of the SIC monitor by making also the GreG system available in addition to the user's commands. The extra code involved is marked by the flag \texttt{!!GREG!!} and can be removed without affecting the user commands.

The list of routines described here is, in current order :  

| USER | The main program |
| LOAD\_USER | The initialization routine (#) |
| EXEC\_USER | To enter Execute Mode. Execute a single SIC/USER valid command (may be a call to a procedure) and returns (#). Switches to Read Mode on an error condition (Interactive session only) or if a PAUSE is encountered. Switches back to Execute Mode with \texttt{CONTINUE} or \texttt{EXIT}. |
| PLAY\_USER | To enter Read Mode with a SIC/USER valid command passed as the first thing to do (#) |
| ENTER\_USER | To enter Read Mode (#). Reads commands on the logical unit 5. Error recovery is provided for interactive sessions only. |
| RUN\_USER | Dispatching program to execute the user commands |
| COM\_1 | Example of interface between a "simple" Fortran subroutine and a SIC command. |
| US\_EXEC | Interface routine for the Library Version |
| US\_ERROR | idem. |

(#) The inclusion of the extra code flagged as \texttt{!!GREG!!} gives also access to the command of \texttt{GREG} in the three routines EXEC\_USER, PLAY\_USER and ENTER\_USER.
In addition, the “true” library mode is accessible by calling US_EXEC, which is described completely in Section 2.6. This is a complete example which really contains all the relevant structure. The rules for building conventional names are quite obvious.

The following source program available in VMS GAG_UTIL:SICUSER.FOR, and UNIX $GAG_REF/help/sic/sicuser.f)

PROGRAM USER
INTEGER ICODE,SIC_GET_FOREIGN,LENC,L,NL,SIC_PURGE
CHARACTER*80 FILE,COMMAND,CHAIN,NAMES*12
CHARACTER BACKSLASH*1
LOGICAL EXIST
*
* Optionally, get the Foreign argument.
  ICODE = SIC_GET_FOREIGN(COMMAND,NL)
  IF (MOD(ICODE,2).EQ.0) CALL SYSEX(IICODE)
* Build the names of the log file.
  NAME = 'user'
  CALL SIC_PARSEF (NAME,FILE,'GAG_LOG:','log')
* Purge older versions of the log file
  ICODE = SIC_PURGE (FILE,1)
  IF (MOD(ICODE,2).EQ.0) CALL SYSEX(IICODE)
*
* Initialize the command monitor with a nice Prompt, and stack capability
  NAME = 'user'
  CALL SIC_OPT(NAME,FILE,.TRUE.)
* Then load all languages
  CALL LOAD_USER
* Demonstration of multi-language capabilities
  CALL LOAD_GREG('LIBRARY') !!GREG!!
* Define a default macro extension
  CHAIN = 'SIC'/BACKSLASH//'SIC EXTENSION .user'
* Execute the "login" macro
  NAME = 'init'
  CALL SIC_PARSEF (NAME,FILE,'GAG_INIT:','.user')
  L = LENC(FILE)
  INQUIRE (FILE=FILE(1:L),EXIST=EXIST)
  IF (EXIST) THEN
    CHAIN = '@' '!'//'FILE(1:L)//''
    CALL EXEC_USER (CHAIN)
  ENDIF
* Activate the monitor
  IF (NL.NE.0) THEN
* First executing the foreign argument if available
  CALL PLAY_USER(COMMAND(1:NL))
ELSE
* Or not
  CALL ENTER_USER
ENDIF
END

SUBROUTINE LOAD_USER
INTEGER MCOM
PARAMETER (MCOM=3)
The example shows how to retrieve a first command from the operating system at run time (SIC\_GET\_FOREIGN), how to define a default macro extension, and how to execute the user initialization macro if it exists. Two types of complete calls to the SIC/USER system are demonstrated.

```plaintext
CHARACTER*12 VOCAB(MCOM)
DATA VOCAB / 'COM1', '/GAG', 'COM2' /
CALL SIC\_LOAD('USER', 'GAG\_HELP:sicuser.hlp', MCOM, VOCAB, 
& '4.0 24-AUG-1984')
END
```

SUBROUTINE EXEC\_USER(BUFFER)
LOGICAL ERROR
CHARACTER(*) BUFFER
CHARACTER LINE*256, COMM*12, LANG*12
INTEGER ICODE, OCODE
*
LINE = BUFFER
ICODE = -1
GOTO 10
*
ENTRY PLAY\_USER(BUFFER)
LINE = BUFFER
ICODE = 2
GOTO 10
*
ENTRY ENTER\_USER
ICODE = 1
GOTO 10
*
* Call the monitor again after completion of every user command.
10 CALL SIC\_RUN (LINE,LANG,COMM,ERROR,ICODE,OCODE)
IF (OCODE.EQ.0) THEN
  IF (LANG.EQ.'USER') THEN
    CALL RUN\_USER (LINE,COMM,ERROR)
  ELSEIF (LANG.EQ.'GREG1') THEN !!GREG!!
    CALL RUN\_GREG1 (LINE,COMM,ERROR) !!GREG!!
  ELSEIF (LANG.EQ.'GREG2') THEN !!GREG!!
    CALL RUN\_GREG2 (LINE,COMM,ERROR) !!GREG!!
  ELSEIF (LANG.EQ.'GTVL') THEN !!GREG!!
    CALL RUN\_GTVL (LINE,COMM,ERROR) !!GREG!!
  IF (COMM.EQ.'CLEAR') CALL GRESET !!GREG!!
  ELSE
    WRITE(6,*) 'Unrecognized Language ',LANG
    ERROR = .TRUE.
  ENDIF
ELSEIF (OCODE.EQ.-1) THEN
* Must return immediately from the Execute Mode,
  RETURN
ELSEIF (OCODE.EQ.1) THEN
* Any other code may be added before returning to the main program
  RETURN
ENDIF
ICODE = 0
```
In the Execute Mode, all SIC capabilities are enabled. A more restrictive Library Mode, in which only the command line interpreter of SIC is used will be shown in Section 2.6

SUBROUTINE RUN_USER(LINE,COMM,ERROR)
CHARACTER*(*) LINE,COMM
LOGICAL ERROR
*
* Call appropriate subroutine according to COMM
  IF (COMM.EQ.'COM1') THEN
    CALL COM1(LINE,ERROR)
  ELSEIF (COMM.EQ.'COM2') THEN
    WRITE (6,*) 'Command COM2 activated'
    CALL COM2(LINE,ERROR)
  ENDIF
END

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

As in this example, it is HIGHLY RECOMMENDED to pass the command buffer LINE by argument and NOT IN A COMMON. This will allow later an easy implementation of a multi-language program if necessary, and the possibility of sharing languages with other users. Use of commons may break these possibilities. It is also recommended to have an “interface” routine for each command which takes all the decoding functions. However, for very simple commands, this may not be necessary and the interface may be in the RUN_USER routine.
2.4 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 END
```

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 END (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_LOAD call, or given explicitly.

2.5 Retrieving Arguments

To properly interface your program, you must know how to retrieve an argument from the command buffer processed by SICIn 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.

<table>
<thead>
<tr>
<th>Argument Type</th>
<th>Routine Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer*4</td>
<td>SIC_I4</td>
</tr>
<tr>
<td>Real*4</td>
<td>SIC_R4</td>
</tr>
<tr>
<td>Real*8</td>
<td>SIC_R8</td>
</tr>
<tr>
<td>Logical*4</td>
<td>SIC_L4</td>
</tr>
<tr>
<td>Character</td>
<td>SIC_CH</td>
</tr>
<tr>
<td>Character</td>
<td>SIC_CE</td>
</tr>
</tbody>
</table>

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, Larg, PRESENT, ERROR)
CALL SIC.KE (LINE, IOPT, IARG, 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 uppercase keywords.

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 2.9.1.

2.6 The Library Version

2.6.1 Creating the Library Interface

Once you have completed the previous operations, an interactive system is available. The system can also be used in batch or within command procedures. It is very easy to make a Fortran callable version of this system by adding only one extra subroutine. Taking again the same example this subroutine will be:

```
FUNCTION US_ERROR(dummy)
*
* Library version of the user program.
* All the user commands are accessible through calls
* in the form:
* CALL US_EXEC ('This_command')
*
* The error status can be checked and reset using the
* function US_ERROR(). If an error exist, the program
* aborts when trying to execute the next command.
* INCLUDE 'inc:errcod.inc'
* The above line defines the FATALE error code used below
LOGICAL ERROR, US_ERROR
CHARACTER(*) BUFFER
CHARACTER LINE*256, COMM*12, LANG*12, MESSAGE*60
INTEGER NL, DUMMY, LENC
* Save the error code from last call
SAVE ERROR
*
* Transmit error status, and reset it
US_ERROR = ERROR
ERROR = .FALSE.
RETURN
```
* ENTRY US_EXEC(BUFFER)
* Check error status, abort if not cleared
  IF (ERROR) THEN
    WRITE(6,*), MESSAGE
    WRITE(6,*) LINE(1:NL)
    CALL SYSEX (FATAL)
  ENDIF
* Copy the command line in argument to the local buffer LINE
* Format it and Analyse it
  LINE = BUFFER
  NL = LENC (LINE)
  CALL SIC_FORMAT (LINE, NL)
  CALL SIC_ANALYSE (COMM, LINE, NL, ERROR)
  IF (ERROR) THEN
    MESSAGE = 'E-US_EXEC, Error Interpreting Line'
    RETURN
  ENDIF
* If your application is multi-language, then make sure that the
  command belongs to your language.
  CALL SIC_LANG (LANG)
  IF (LANG .NE. 'USER') THEN
    ERROR = .TRUE.
    MESSAGE = 'E-US_EXEC, Language Mismatch Line'
    RETURN
  ENDIF
* Execute it
  CALL RUN_USER (LINE, COMM, ERROR)
  IF (ERROR) THEN
    MESSAGE 'E-US_EXEC, Error Executing Line'
  ENDIF
END

A program can then be written just by typing the same commands as in interactive. Of course, this is not always possible, but you should be able to provide additional routines to make the formatting of the line, or give a more direct argument transmission. Note that all errors are absolutely fatal in this Library Mode, and must be handled by the calling program. This is in contrast with the EXECJSER routine which provided a SIC like error recovery, but at the expense of a compulsory interactive use of the program.

2.6.2 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. This means that the commands are accessible only through a US_EXEC like program and not through SIC monitor capabilities. 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 null help specification:

SIC_LOAD (LANG, ' ', MCOM, VOCAB, VERSION)
2.6.3 Reserved logical unit numbers: VMS

SIC has the following usage of Fortran logical unit numbers:

5    Input of Commands, shareable
6    Output of messages, shareable

Units 5 and 6 are respectively accessed via usual READ and WRITE instructions. In addition, the terminal is accessed via QIO on an assigned channel for keypad line editing. The associated logical names are FOR005 and FOR006 and the default assignments provided by the system VMS are:

ASSIGN SYS$INPUT FOR005
ASSIGN SYS$OUTPUT FOR006

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\_MPR, SIC\_MPRN routine). If you mix READ(5,format) and WRITE(6,format) instructions with TYPE, ACCEPT, READ(*,format) or WRITE(*,format) instructions, you might experience some problems to properly re-assign input or output. As a hint for this, remember the default logical names for Input/Output on VAX/VMS:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Fortran Log Name</th>
<th>System Log Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(mnn,f) list</td>
<td>FOR_mnn</td>
<td>SYS$INPUT</td>
</tr>
<tr>
<td>READ(*,f) list</td>
<td>FOR_READ</td>
<td>SYS$INPUT</td>
</tr>
<tr>
<td>READ(f) list</td>
<td>FOR_READ</td>
<td>SYS$INPUT</td>
</tr>
<tr>
<td>ACCEPT f,list</td>
<td>FOR_ACCEPT</td>
<td>SYS$INPUT</td>
</tr>
<tr>
<td>WRITE(mnn,f) list</td>
<td>FOR_mnn</td>
<td>SYS$OUTPUT</td>
</tr>
<tr>
<td>WRITE(*,f) list</td>
<td>FOR_PRINT</td>
<td>SYS$OUTPUT</td>
</tr>
<tr>
<td>PRINT f,list</td>
<td>FOR_PRINT</td>
<td>SYS$OUTPUT</td>
</tr>
<tr>
<td>TYPE f,list</td>
<td>FOR_TYPE</td>
<td>SYS$OUTPUT</td>
</tr>
</tbody>
</table>

The monitor uses additional logical units for the LOG file, for opening temporary files and for active macro’s. These units are reserved dynamically by calls to the VAX-VMS appropriate services and do not interfere with any Fortran unit numbers used within the user program, except that a program may use too many logical units.

2.6.4 Reserved logical unit numbers: UNIX

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. 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\_MPR, SIC\_MPRN 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 is recommended to printout messages.

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.

2.6.5 Linking on VMS

In order to avoid an overhead in program size and link time, the SIC system takes advantage of the shareable library facilities. All the SIC system is contained in the module SIC\_SHR available in the library GAG\_ROUT:[SHARED]GAGSHARE.OLB.
The access to this library is automatic on some computers. Issue the command
\$ SHOW LOGICAL $LNK$LIB*

To know if automatic linking is enabled.

If not enabled, you may either make an explicit reference in the link as:
\$ LINK PROG1,PROG2,...,GAG_ROOT:[SHARED]GAGSHARE/LIB

Either add the following assignment in your LOGIN.COM for automatic linking to this library:
\$ ASSIGN GAG_ROOT:[SHARED]GAGSHARE.0LB $LNK$LIBRARY

Note that is you wish to use the GreG library together with SIC you must also reference the library
GAG_ROOT:[SHARED]GREGLIB.0LB as described in the GreG manual.

2.6.6 Linking on UNIX

On Unix system, the location of the GAG software environment is normally defined by the $GAG_ROOT
evironment variable. If not, ask your system manager to add the definition of $GAG_ROOT in a system-wide
"login" script.

Several libraries are needed to run the SIC system. Libraries may be shareable or simple archive; the
operating system will normally decide which one to use. All libraries are in the directory $GAG_ROOT/lib.
The link command should look like (e.g. for Fortran program on HP-UX systems):

fort77 -L$GAG_ROOT/lib -o prog prog.o -lsic -llmage -lgag -lrary

Other compilers (such as cc, the C compiler) will have similar options, but note however that the Fortran
libraries will be needed in the link. Beware that the order of the libraries is significant. For systems not
supporting the -L option, you will have to explicitly mention the full names of the libraries.

2.7 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 SAVED (or appear in a SAVED COMMON) to prevent
the compiler to allocate it on the stack.

2.7.1 Definition

There are 5 variable declaration routines following the same calling conventions

<table>
<thead>
<tr>
<th>Argument Data Type</th>
<th>Name of Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL*8</td>
<td>SIC_DEF_DBLE</td>
</tr>
<tr>
<td>REAL*4</td>
<td>SIC_DEF_REAL</td>
</tr>
<tr>
<td>INTEGER*4</td>
<td>SIC_DEF_INT</td>
</tr>
<tr>
<td>CHARACTER*(*)</td>
<td>SIC_DEF_CHAR</td>
</tr>
<tr>
<td>LOGICAL*4</td>
<td>SIC_DEF_LOGI</td>
</tr>
</tbody>
</table>

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, and INTE, NDIM being the number of dimensions,
and DIM the dimensions of the array 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. Variable names must be at most 15 characters, upper case only, and beginning with a letter. Special characters must be avoided, except the "_" (underscore) sign.

2.7.2 Assignment and Examination

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

<table>
<thead>
<tr>
<th>Argument Data Type</th>
<th>Assignment</th>
<th>Examination Subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL*8</td>
<td>SIC_LDTDBLE</td>
<td>SIC_GETDBLE</td>
</tr>
<tr>
<td>REAL*4</td>
<td>SIC_LDAREAL</td>
<td>SIC_GETREAL</td>
</tr>
<tr>
<td>INTEGER*4</td>
<td>SIC_LDINTER</td>
<td>SIC_GETINTER</td>
</tr>
<tr>
<td>LOGICAL*4</td>
<td>SIC_LDLLOGI</td>
<td>SIC_GETLOGI</td>
</tr>
<tr>
<td>CHARACTER(*)</td>
<td>SIC_LDTCHAR</td>
<td>SIC_GETCHAR</td>
</tr>
</tbody>
</table>

The calling syntax is the following:

CALL SIC_LDN_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 is the variable name (general syntax A[i,j] allowed)
- DESC is the descriptor of the variable, an integer array of dimension 9 containing

  DESC(1) Variable type, with READONLY code. The true variable type is VAR_TYPE(DESC)
  Valid values are
  * 0 < VAR_TYPE(DESC) : Character string of length VAR_TYPE(DESC)
  * VAR_TYPE(DESC) < 0 may have the value
    FMT_R4 for REAL*4 variable
    FMT_R8 for REAL*8 variable
    FMT_I4 for INTEGER*4 variable
    FMT_L for LOGICAL*4 variable
    FMT_BY for BYTE variable
    FMT_C4 for COMPLEX variables
  * If VAR_TYPE(DESC) is not equal to DESC(1), the variable is READONLY.

  DESC(2) Variable address
  DESC(3) Number of dimensions
  DESC(4) = DESC(7) Dimensions
2 SIC PROGRAMMING MANUAL

DESC(8) Size of allocation (words)
DESC(9) Origin of variable
   0 program defined
   < 0 user defined
   > 0 image
- FOUND is a logical indicating whether the specified variable
  exist or not.

The parameters FMT_R4, FMT_R8, FMT_I4, FMT_L, FMT_BY and FMT_C4 are defined in the file
inc:format.inc.
Two other routines may be required for variable handling:
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 integer arrays of dimension 9. DESC should have been obtained by a previous call to
SIC_DESCRIPTOR.
Once used, the incarnation may be deleted using routine SIC_VOLATILE

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

2.7.3 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.

2.7.4 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.
2.8 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)
```

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

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)
```

2.9 SIC Callable Routines

2.9.1 Monitor interface routines

**Available subroutines:**
- **SIC_LOAD**
- **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 (**SIC_LOAD** and **SIC_RUN**), others are simply provided for additional capabilities. These routines have a standard **SIC** like name and are fully described in this section.

**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 **SIC**. No need to initialize it when **ICODE** = 0 or 1, but must be initialized if **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.

SIC_LOAD (LANG, HELP, NCOM, CCOM, VERSION)
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*12 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.

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_LOAD.

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 _i> 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_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 SICOPT. 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 2.2). 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_INTERSTATE ( )

This subroutine returns .TRUE. if the session is interactive, .FALSE. otherwise. This is the case if SIC is ran during a batch process, or if the logical unit FOR005 is assigned to a file (either directly or through an assignment of SYS$INPUT).
### 2.9.2 SIC Arguments Retrieving Routines

**Available subroutines:**

<table>
<thead>
<tr>
<th>SIC_L4</th>
<th>SIC_I4</th>
<th>SIC_R4</th>
<th>SIC_R8</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIC_CH</td>
<td>SICKE</td>
<td>SICNEXT</td>
<td></td>
</tr>
</tbody>
</table>

**Available functions:**

<table>
<thead>
<tr>
<th>SIC_PRESENT</th>
<th>SIC_LEN</th>
<th>SIC_NARG</th>
<th>SIC_START</th>
</tr>
</thead>
</table>

There are 5 retrieving routines following similar calling conventions:

```
Argument Data Type     Name of Subroutine
------------------------------------------------
REAL*8                 SIC_R8
REAL*4                 SIC_R4
INTEGER*4              SIC_I4
LOGICAL*4              SIC_L4
CHARACTER*(*)          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 occurred.

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,MVC,VOC,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.

• IVQC Integer
  The keyword number in the vocabulary.

• MVQC Integer
  The number of keywords in the vocabulary.

• VOCAB Character*(*) (MVQC)
  The vocabulary.

• ERROR Logical
  An error flag set if the input name was ambiguous, invalid or not found in the vocabulary.

INTEGER FUNCTION SICLEN(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 SICSTART(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 SICLEN for special processing of
  arguments, such as sexagesimal decoding.

SUBROUTINE SICNEXT(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_REAL or SIC_COMPLEX, logical value using routine SIC_LOGICAL, or used as character constant.

2.9.3 Command Line Interpreter 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*12, Command name (returned)
• LINE Character*(*), 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 interpreter.
  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).
2.9.4 All Purpose General Subroutines

Available subroutines:

- SIC_LOWER
- SIC_PARSER
- SIC_UPPER
- SIC_WPRN

Available functions

- SIC_GETVM
- FREE_VM
- LENC

These routines are of more general use than SIC itself.

INTEGER FUNCTION SIC_GETVM(N32, ADDR)
SIC_GETVM allocates virtual memory for N32 32-bit words, and returns the allocated address space into variable ADDR. SIC_GETVM is set to 1 in case of success, to other otherwise.

SUBROUTINE FREE_VM(N32, ADDR)
FREE_VM can be used to free the corresponding address space when no longer needed.

SUBROUTINE SIC_LOWER(LINE) - SIC_UPPER(LINE)
Converts the string LINE to lower or upper case letters respectively.

SUBROUTINE SIC_PARSER (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

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.

2.9.5 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.
3 GreG Programming Manual

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 EXEC3 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 optimised 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.

3.1 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_SETLUN.

• 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_CTRL 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_EXEC1 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.

3.2 Basic Routines

1. LOAD_GREG (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.DPT. 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 3 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. ENTER_GREG enter GreG interactive mode. The user obtains control over GreG exactly as if he
   was running the GreG utility. This subroutine can only be used if GreG has been initialized in
   "Interactive" mode.

5. EXEC_GREG(LINE)
   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.
6. **PLAY_GREG(LINE)**
   execute a SIC or **GreG** command and then leave **GreG** in interactive mode. This call is thus more or less equivalent to successive calls to **EXEC_GREG** and **ENTER_GREG** (there is a minor difference in case of error). Control returns to the calling program after **EXIT**. “Interactive” mode is required.

*Only **GR.EXEC** (or its variant) and **GR.ERROR** are allowed in Library Mode.*

### 3.3 Linking

#### 3.3.1 UNIX systems

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 -lgftlang -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.

#### 3.3.2 VMS systems

To access **GreG** from your Fortran Program, you need to link to

First library : 
   
   GAG_ROOT: [SHARED]GREGLIB/LIB

Second library : 
   
   GAG_ROOT: [SHARED]GAGSHARE/LIB

The access to these libraries is automatic on some machines. On other sites, issue the command `$ SHOW LOGICAL LNK$LIB*` to know if automatic linking is enabled. If not enabled, you may either make an explicit reference in the link as :

```bash
$ LINK PROG1,PROG2,...,-
   GAG_ROOT:[SHARED]GREGLIB/LIB,GAGSHARE/LIB
```

or add the following assignments in your LOGIN.COM file for automatic linking to these libraries :

```bash
$ ASSIGN GAG_ROOT:[SHARED]GREGLIB.OLB LNK$LIBRARY
$ ASSIGN GAG_ROOT:[SHARED]GAGSHARE.OLB LNK$LIBRARY_1
```

Note however that the GREGLIB library must be specified before the GAGSHARE library in the link command. Be careful if you have other shareable libraries in your link operation to have all these libraries placed as the last ones. As most of the code is shareable, linking to **GreG** is fast and does not produce a large code overhead. Also, you will get the minor updates of **GreG** automatically without need to relink your program.

However you may need to relink your program for major updates. In this case you will receive a message at run-time stating that the version number of the updated shared images (**GREGSHR, CHARSHR, CONSCHR, GTSHR, IMASHR, GAGSRH** or **SICSHR**) no longer matches the version number at link-time and you will not be able to run your program without relinking it. If this happens to you, please consult the latest description of the **GreG** manual to see whether any calling sequence might have been modified. If not, linking again will be sufficient.
3.4 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 the log file will be written on the logical unit LOG, (unless you call SIC_DP, see the SIC manual). If you have not assigned the logical name LOG to a file name, each run of your program will create a new version of the file LOG.DAT in your current directory. No automatic purge of the log file will be made when your program exits.

3.5 Example

A simple and typical example is available in the Fortran file GAG_UTIL:GREATTEST.FOR. Just print it. If it is close to your application, you may copy it into your directory and adapt it to your wishes.

3.6 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*4 for the GR4_ version, Real*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).

3.6.1 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*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.

3.6.2 GR4_GET - GR8_GET

**GR4_GET(NAME, NXY, ARRAY)**

This subroutine is the reverse of the GR4(G) 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*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

3.6.3 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*N 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)

If R is Real^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^8, virtual memory is allocated to create a Real^4 array of same dimensions.

3.6.4 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.

3.7 Immediate Routines

The following routines do not use the SIC command line interpreter. They are essentially internal GreG routines made available to the user, and are thus very efficient to use. As above, they have a Real^4 and a Real^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

3.7.1 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.

3.7.2 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.

3.7.3 DRAW - RELOCATE

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

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

3.7.4 GDRAW - GRELOCATE

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

Basic pen down or up movement in Physical coordinates (Real^4 values). There is no Real^8 version.
3.7.5 **GR4\_PHYS\_USER - GR8\_PHYS\_USER**

GR4\_PHYS\_USER (XP, YP, XU, YU, NXY)
- Convert Physical coordinates (XP, YP always Real*4 values) to User coordinates (XU, YU, REAL values). NXY is the number of values.

3.7.6 **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*4 values). NXY is the number of values.

3.7.7 **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

3.7.8 **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

3.7.9 **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

3.7.10 **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

3.7.11 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

3.7.12 GR8_BLANKING

GR8_BLANKING (BVAL, EVAL)

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

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

3.7.13 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 GONOMONIC, 2 for ORTHOGRAPHIC, 3 for AZIMUTHAL, 4 for STEREGRAPHIC, 5 for LAMBERT cylindrical, 6 for AITOFF equal area and 7 for RADIAL (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

3.7.14 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

3.7.15 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

3.7.16 GR8_TRI

GR8_TRI(X,INDEX,N,*)
Sorting program that uses a quicksort algorithm. Applies for an input array of Real*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    R8(*)  Unsorted/Sorted array  Input/Output
INDEX I(*)  Integer array of sorted indexes  Output
N    I  Length of arrays  Input
*    Label  Error return

3.7.17 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    R8(*)  Unsorted/Sorted array  Input/Output
XSORT R8(*)  Sorted array  Work space
INDEX I(*)  Integer array of sorted indexes  Input
           (obtained by GR8_TRI)
N    I  Length of arrays  Input

3.7.18 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.

3.8 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:

<table>
<thead>
<tr>
<th>XU</th>
<th>REAL*8</th>
<th>X User coordinates</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>YU</td>
<td>REAL*8</td>
<td>Y User coordinates</td>
<td>Output</td>
</tr>
<tr>
<td>X4</td>
<td>REAL*4</td>
<td>X Plot coordinates</td>
<td>Output</td>
</tr>
<tr>
<td>Y4</td>
<td>REAL*4</td>
<td>Y Plot coordinates</td>
<td>Output</td>
</tr>
<tr>
<td>CODE</td>
<td>CHARACTER*1</td>
<td>Character struck</td>
<td>Output</td>
</tr>
</tbody>
</table>

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

For X-Window terminals, button 1 returns ~, 2 returns & and 3 returns *.  

3.9 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 transmits the command to GR_EXEC.

The conventional type of arguments are

<table>
<thead>
<tr>
<th>NAME</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARG, ARG1, ...</td>
<td>REAL*8</td>
</tr>
<tr>
<td>IARG, IARG1, ...</td>
<td>INTEGER*4</td>
</tr>
<tr>
<td>NARG</td>
<td>INTEGER*4</td>
</tr>
</tbody>
</table>

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_JLABE(’A little toy’).

Example: CALL GR_JDRAW(’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 preceded 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 GREGL1 language, the routines are

AXIS Name [A1 A2] [/LOCATION X Y] [/TICK Orientation [BIG SMALL]]

[/LABEL Position] [/NOLOG] [/ABSOLUTE]

SUBROUTINE GR_AXIS(NAME, NARG, ARG1, ARG2) ! 28-Sep-1986
ENTRY GR_AXIS_LOC(NARG, ARG1, ARG2)
ENTRY GR_AXIS_TICK(NAME, NARG, ARG1, ARG2)
ENTRY GR_AXIS_LABE(NAME)
ENTRY GR_AXIS_LOG
ENTRY GR_AXIS_NULO
ENTRY GR_AXIS_ABSO

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

COLUMN [X Hx] [Y Ny] [Z Nz] [/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('GREG\COLUMN ' /NAME)
or GR_EXEC('COLUMN ' /NAME)

CONNECT [/BLANKING Bval Eval]
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
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(ARG,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 ...').

SUBROUTINE GR_PEN(NARG,IARG)  ! 28-Sep-1986
ENTRY GR_PEN_DASH(IARG)
ENTRY GR_PEN_WEIG(IARG)
ENTRY GR_PEN_CULO(IARG)
ENTRY GR_PEN_DEFA

POINTS [Size] [/BLANKING Eval Eval]
SUBROUTINE GR_POINT(NARG,ARG1)
SUBROUTINE GR_POINT_BLAN(NARG,ARG1,ARG2)

RULE [X] [Y] [/MAJOR [/MINOR]
SUBROUTINE GR_RULE(NAME)  ! 28-Sep-1986
ENTRY GR_RULE_MAJ0
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
SUBROUTINE GR_TICK(NARG,ARG1,ARG2,ARG3,ARG4)  ! 28-Sep-1986

The following commands correspond to the GREG2\Language.

EXTREMA [/BLANKING Eval Eval] [/PLOT]
SUBROUTINE GR_EXTR
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)

[\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.
4 Task Programming Manual

4.1 General Outline and Data Structure of Images

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 recognised: VAX format, IEEE format (DEC DS Workstation) and EEEI format (Sun SparcStation, IBM RS-6000 series, etc...).

The images have the following organisation:

- Images are basically direct access files with a 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 header starts with 'GILDAS', followed by one character which indicates the type of integer and floating point number representation

- Underscore "_" for VAX machines
- Dot "." for swapped IEEE machines (IBM RS-6000, SUN Sparc, etc...)
- Minus "-" for non-swapped IEEE machines (DEC DS series)

and extra text indicating the type of information in the image:

- **IMAGE** for all images (up to 4-D arrays)
- **UVDAT** for tables of unsorted, unprecessed UV data
- **UVSOR** for tables of sorted, precessed UV data

On computers which support such possibilities (VMS systems), GILDAS makes extensive use of memory mapping to access the images: instead of being read into memory through standard I/O techniques, the disk file containing an image is directly made part of the virtual memory accessed by the program, through the use of the appropriate system routines. Physical I/O between this virtual memory and the physical memory is then later handled by the system memory management.

On other computers, the image is copied into memory using standard FORTRAN direct I/O. Provided adequate programming by the application programmer, only the necessary part of the image is read or mapped. This is a totally transparent process for the application programmer, who will always consider the image as a FORTRAN array in virtual memory.

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.
4.2 Fortran-90 access to images

Fortran-90 allows definition of data structures (derived types) which are convenient to handle images. The Fortran-90 module `image_def` defines two derived type corresponding to GILDAS images: the type GILDAS and the type SIC.

```fortran
MODULE IMAGE_DEF
  INTEGER CODE_READ_DATA
  INTEGER CODE_READ_HEADER
  INTEGER CODE_UPDATE_HEADER
  INTEGER CODE_WRITE_HEADER
  INTEGER CODE_WRITE_DATA
  INTEGER CODE_READ_IMAGE
  INTEGER CODE_FREE_IMAGE
  INTEGER CODE_CREATE_IMAGE
  !
  PARAMETER (CODE_READ_DATA=-1)
  PARAMETER (CODE_READ_HEADER=-2)
  PARAMETER (CODE_UPDATE_HEADER=-3)
  PARAMETER (CODE_WRITE_HEADER=-4)
  PARAMETER (CODE_WRITE_DATA=-5)
  PARAMETER (CODE_READ_IMAGE=-6)
  PARAMETER (CODE_FREE_IMAGE=-7)
  PARAMETER (CODE_CREATE_IMAGE=-8)
  !
  INTEGER FATALE
  PARAMETER (FATALE=44)
  !
  /XIPAR/
  TYPE LOCATION
    SEQUENCE
      INTEGER*4 AL64
      INTEGER*4 SIZE
      INTEGER(KIND=4) ADDR
      INTEGER ISLO
      INTEGER MSL0
      LOGICAL READ
      LOGICAL GETVM
  END TYPE LOCATION
  ! /XPAR/
  TYPE STRINGS
    SEQUENCE
      CHARACTER*12 TYPE
      CHARACTER*12 UNIT       ! 56
      CHARACTER*12 CODE(4)     ! 59
      CHARACTER*12 SYST        ! 71
      CHARACTER*12 NAME        ! 75,76,77
      CHARACTER*12 LINE
  END TYPE STRINGS
  ! /XPAR/
  TYPE GILDAS_HEADER
    SEQUENCE
      INTEGER*4 ITYP(3)        ! 1
      INTEGER*4 FORM           ! 4
      INTEGER*4 NVB            ! 5
```

INTEGER*4 FILL(5) ! 6
INTEGER*4 GENE ! 11
INTEGER*4 NDIM ! 12
INTEGER*4 DIM(4) ! 13
REAL*8 REF1 ! 17
REAL*8 VAL1 ! 19
REAL*8 INC1 ! 21
REAL*8 REF2 ! 23
REAL*8 VAL2 ! 25
REAL*8 INC2 ! 27
REAL*8 REF3 ! 29
REAL*8 VAL3 ! 31
REAL*8 INC3 ! 33
REAL*8 REF4 ! 35
REAL*8 VAL4 ! 37
REAL*8 INC4 ! 39
INTEGER*4 BLAN ! 41
REAL*4 EVAL ! 42
REAL*4 EVAL ! 43
INTEGER*4 EXTR ! 44
REAL*4 RMIN ! 45
REAL*4 RMAX ! 46
INTEGER*4 MIN1 ! 47
INTEGER*4 MAX1 ! 48
INTEGER*4 MIN2 ! 49
INTEGER*4 MAX2 ! 50
INTEGER*4 MIN3 ! 51
INTEGER*4 MAX3 ! 52
INTEGER*4 MIN4 ! 53
INTEGER*4 MAX4 ! 54
INTEGER*4 DESC ! 55
INTEGER*4 IUNI(3) ! 56
INTEGER*4 ICOD(3,4) ! 59
INTEGER*4 ISYS(3) ! 71
INTEGER*4 DUM1 ! Void
INTEGER*4 POS1 ! 74
INTEGER*4 ISOU(3) ! 75
REAL*8 RA ! 78
REAL*8 DEC ! 80
REAL*8 LII ! 82
REAL*8 BII ! 84
REAL*4 EPQC ! 86
INTEGER*4 DUM2 ! Void
INTEGER*4 PROJ ! 87
INTEGER*4 PTYP ! 88
REAL*8 A0 ! 89
REAL*8 DO ! 91
REAL*8 PANG ! 93
INTEGER*4 XAXI ! 95
INTEGER*4 YAXI ! 96

INTEGER*4 SPEC ! 97
INTEGER*4 ILIN(3) ! 98
REAL*8 PRES ! 101
REAL*8 FIMA ! 103
REAL*8 FREQ ! 105
REAL*4 VRES ! 107
REAL*4 VOFF ! 108
INTEGER*4 FAXI ! 109

INTEGER*4 RESO ! 110
REAL*4 MAJO ! 111
REAL*4 MINO ! 112
REAL*4 POSA ! 113

INTEGER*4 SIGM ! 114
REAL*4 NOISE ! 115
REAL*4 RMS ! 116
END TYPE GILDAS_HEADER

TYPE SIC_HEADER
SEQUENCE
INTEGER*4 ITYP(3) ! 1
INTEGER*4 FORM ! 4
INTEGER*4 NVB ! 5
INTEGER*4 FILL(5) ! 6
INTEGER*4 GENE ! 11
INTEGER*4 NDIM ! 12
INTEGER*4 DIM(4) ! 13
REAL*8 CONVERT(3, 4) ! 17-40

INTEGER*4 BLAN ! 41
REAL*4 BLANK(2) ! 42-43

INTEGER*4 EXTREMA ! 44
REAL*4 MIN ! 45
REAL*4 MAX ! 46
INTEGER*4 WHERE(2, 4) ! 47-54

INTEGER*4 DESC ! 55
INTEGER*4 IUNI(3) ! 56 Unit
INTEGER*4 ICOD(3, 4) ! 59 Unit 1, 2, 3, 4
INTEGER*4 ISYS(3) ! 71 System
INTEGER*4 DUM1 ! Void

INTEGER*4 POSI ! 74
INTEGER*4 ISQU(3) ! 75 Source
REAL*8 RA ! 78
REAL*8 DEC ! 80
REAL*8 LII ! 82
REAL*8 BII ! 84
REAL*4 EPOC ! 86
INTEGER*4 DUM2 ! Void
! INTEGER*4 PROJ      ! 87
INTEGER*4 PTYP       ! 88
REAL*8 A0            ! 89
REAL*8 DO            ! 91
REAL*8 ANGLE         ! 93
INTEGER*4 X_AXIS     ! 95
INTEGER*4 Y_AXIS     ! 96

INTEGER*4 SPEC       ! 97
INTEGER*4 ILIN(3)    ! 98 Line
REAL*8 FREQRES       ! 101
REAL*8 FREOFF        ! 103
REAL*8 BRESTFRE      ! 105
REAL*4 VELRES        ! 107
REAL*4 VELOFF        ! 108
INTEGER*4 F_AXIS     ! 109

INTEGER*4 BEAM       ! 110
REAL*4 MAJOR         ! 111
REAL*4 MINOR         ! 112
REAL*4 PA            ! 113

INTEGER*4 SIGMA      ! 114
REAL*4 NOISE         ! 115
REAL*4 RMS           ! 116
END TYPE SIC_HEADER
!
! Gildas X,Y,Z version

TYPE GILDAS

SEQUENCE

CHARACTER*256 FILE   ! File name
TYPE (STRINGS) :: CHAR
TYPE (LOCATION) :: LOCA
TYPE (GILDAS_HEADER) :: GIL
INTEGER*4 BLK(4)
INTEGER*4 TRC(4)
INTEGER*4 HEADER     ! Defined / Undefined
INTEGER*4 STATUS     ! Last error code
REAL, POINTER :: R1D(:)
REAL(KIND=8), POINTER :: D1D(:)
INTEGER, POINTER :: I1D(:)
REAL, POINTER :: R2D(:,;)
REAL(KIND=8), POINTER :: D2D(:,;)
INTEGER, POINTER :: I2D(:,;)
REAL, POINTER :: R3D(:,;:)
REAL(KIND=8), POINTER :: D3D(:,;:)
INTEGER, POINTER :: I3D(:,;:)
REAL, POINTER :: R4D(:,;:;)
REAL(KIND=8), POINTER :: D4D(:,;:;)
INTEGER, POINTER :: I4D(:,;:;)

END TYPE GILDAS
!

! SIC Variable version
TYPE SIC
   SEQUENCE
   CHARACTER*256 FILE           ! File name
   TYPE (STRINGS) :: CHAR
   TYPE (LOCATION) :: LOCA
   TYPE (SIC_HEADER) :: SIC
   INTEGER*4 BLC(4)
   INTEGER*4 TRC(4)
   INTEGER*4 HEADER
   INTEGER*4 STATUS
   REAL, POINTER :: R1D(:)
   REAL(KIND=8), POINTER :: D1D(:)
   INTEGER, POINTER :: I1D(:)
   REAL, POINTER :: R2D(:,:)
   REAL(KIND=8), POINTER :: D2D(:,:)
   INTEGER, POINTER :: I2D(:,:)
   REAL, POINTER :: R3D(:,,:,:)
   REAL(KIND=8), POINTER :: D3D(:,,:,:)
   INTEGER, POINTER :: I3D(:,,:,:)
   REAL, POINTER :: R4D(:,,:,:,:)
   REAL(KIND=8), POINTER :: D4D(:,,:,:,:)
   INTEGER, POINTER :: I4D(:,,:,:,:)
END TYPE SIC
END MODULE IMAGE_DEF

Types GILDAS and SIC can be used indifferently to represent the same object. Type GILDAS mimics the old Fortran-77 commons which were used to handle image headers, while type SIC mimics the SIC header variables.

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
   LOGICAL ERROR
   INTEGER IER
   CHARACTER*32 NAME1,NAME2
   ! TYPE (GILDAS) :: INPUT_IMAGE, OUTPUT_IMAGE     ! 2
   REAL, ALLOCATABLE :: INPUT(:,:,:), OUTPUT(,:,:,:) ! 3
!
   CALL GILDAS_OPEN
   CALL GILDAS_CHAR('INPUT$',NAME1)
   CALL GILDAS_CHAR('OUTPUT$',NAME2)
   CALL GILDAS_CLOSE
!
   CALL GILDAS_NULL(INPUT_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')
   ENDIF
   STOP
END
   ALLOCATE(INPUT(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')
STOP
ENDIF
CALL GDF_READ_DATA (INPUT_IMAGE, DINPUT, ERROR)  ! 8
IF (ERROR) THEN
   CALL GAGOUT('E-IMAGE_EXAMPLE, Error reading input file')
STOP
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%GILDIM - 3  ! 11
OUTPUT_IMAGE=GILDIM(1) - INPUT_IMAGE=GILDIM(1)  ! 11
OUTPUT_IMAGE=GILDIM(2) - INPUT_IMAGE=GILDIM(2)  ! 11
OUTPUT_IMAGE=GILDIM(3) - 4  ! 11
ALLOCATE(DOUTPUT(OUTPUT_IMAGE=GILDIM(1), &
   OUTPUT_IMAGE=GILDIM(2),OUTPUT_IMAGE=GILDIM(3), &
STAT-IER)  ! 12
IF (IER.NE.0) THEN
   CALL GAGOUT('E-IMAGE_EXAMPLE, Error allocating memory')
STOP
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')
STOP
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. Reset the input image header to “default”
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)**
  Reset a header to default values

- **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)**
  Tranpose a header according to given transposition code. This routine transposes the axes information, but not the data.

### 4.3 Obsolescent Fortran-77 access routines

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_WHSEC**
- 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**
- preparation of the header using e.g. **GDF_WHSEC**
- creation of the image to the slot using **GDF_WRIS**. 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**
4.3.1 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_FRIS 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).

4.3.2 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, REd 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_SEMS.

SUBROUTINE GDF_GRIS (IS,GTYPE,NNAME,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_WMSEC) before, and is written on the new file header.
SUBROUTINE GDF_EXIS (IS, GTYPE, NAME, FORM, SIZE, ERROR)

GDF E xtend 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 R ead 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 W rite 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.

4.3.3 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
Arguments:
 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
GDF_JEMS 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_JEMS (MS, ERROR)
```

GDF_Free Memory Slot

GDF_JEMS 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_JEMS 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.

### 4.3.4 Image Header Access

Each image slot has an associated buffer area, from which header information can be retrieved. This buffer, named HEADER(*,IS) for slot IS, is an array of size 128 words, containing an copy of the file header, but converted to the machine-dependent number format. Subroutines are available to read (or write) the header section from (to) these internal buffers: GDF_BHSEC and GDF_WHSEC.

To provide pre-defined data structures with the appropriate header item type, 3 named commons called X, Y or Z are described in include files inc:start.inc, where "a" is the common "name". Routines are available to copy headers from one common to another (GDF.Copy), and to read header from (or copy header to) the internal array HEADER (GDF_Head, GDF_Write).

Each named common contains a variable to define one memory slot, a MSLO, but more memory slots can be used simultaneously by declaring other variables in the application program. Two routines can be used to communicate between the "new" Fortran-90 header structures and the "old" Fortran-77 named commons:

```
SUBROUTINE GDF_HEADER_TO_COMMON (IMAG, COM)
SUBROUTINE GDF_COMMON_TO_HEADER (IMAG, COM)
```

where IMAGE is a header structure, and COM is 'X', 'Y' or 'Z'.

**Named Common Description** The named commons have the following structure, given here for X common. Replace X by Y or Z respectively for the others. Note that each named common is in fact made of 3 different FORTRAN commons, to handle different types of information. Two representations of the character strings exist, one in XCPAR as character variables, one in XPAR as integer arrays. The application program should deal only with the XCPAR representation.
INTEGER MXPAR, MXIPAR, MXPAR
PARAMETER (MXPAR=128) ! 128 header integers
PARAMETER (MXIPAR=5) ! 5 Extra integers
PARAMETER (MXPAR=9) ! 9 character*12 strings
CHARACTER*120 X_FILE ! File name
CHARACTER*12 X_TYPE ! Image type
CHARACTER*12 X_UNIT ! Image unit
CHARACTER*12 X_CODE(4) ! Axis type
CHARACTER*12 X_SYST ! System type
CHARACTER*12 X_NAME ! Source name
CHARACTER*12 X_LINE ! Line name
COMMON /XPAR/ X_FILE,
   X_TYPE,X_UNIT,X_CODE,X_SYST,X_NAME,X_LINE
INTEGER X_SIZE ! image size
INTEGER X_ADDR ! start map address
INTEGER X_ISLO ! Image slot
INTEGER X_MSL0 ! Memory Slot
LOGICAL X_READ ! Read Only status
COMMON /XPAR/ X_SIZE,X_ADDR,X_ISLO,X_MSL0,X_READ
INTEGER X_BUFF(MXPAR)
REAL*8 X_CONV(3,4)
EQUIVALENCE (X_CONV(1,1), X_REF1)
EQUIVALENCE (X_ITYP, X_BUFF)
*
*/XPAR/
INTEGER X_ityp(3) ! Image Type TYPE
INTEGER*4 X_form ! Image Format FORM
INTEGER*4 X_nvbl ! Number of blocks of image
INTEGER*4 X_fill(5) ! Reserved space
*
INTEGER X_GENE ! Size of GENERal section
INTEGER X_NDIM ! Number of dimensions
INTEGER X_DIM(4) ! Dimensions
REAL*8 X_REF1 ! Reference Pixel for First Axis
REAL*8 X_VEC1 ! Value at Reference Pixel
REAL*8 X_INC1 ! Increment per Pixel
REAL*8 X_REF2 ! As above for Second Axis
REAL*8 X_VEC2
REAL*8 X_REF3
REAL*8 X_VEC3
REAL*8 X_REF4
REAL*8 X_VEC4
REAL*8 X_INC4
*
INTEGER X_BLAN ! Length of Blanking Section
REAL X_BVAL ! Blanking Value
REAL X_EVAL ! Tolerance on Blanking
*
INTEGER X_EXTR ! Length of Extrema Section
REAL X_MIN ! Minimum
REAL X_MAX ! Maximum
INTEGER X_MIN1 ! Pixel of Minimum (first axis)
INTEGER X_MAX1  ! " of Maximum
INTEGER X_MIN2  ! " Minimum (second axis)
INTEGER X_MAX2
INTEGER X_MIN3
INTEGER X_MAX3
INTEGER X_MIN4
INTEGER X_MAX4

* INTEGER X_DESC   ! Description section
INTEGER X_IUNI(3) ! Unit Name X_UNIT
INTEGER X_ICOD(3,4) ! Axis Name X_CODE(4)
INTEGER X_ISYS(3) ! Coordinate System X_SYSY

* INTEGER X_POSI   ! Position section
INTEGER X_SOUR(3) ! Source Name X SOUR
REAL*8 X_RA       ! Right Ascension (Radians)
REAL*8 X_DEC      ! Declination (Radians)
REAL*8 X_LII      ! Galactic Longitude (Radians)
REAL*8 X_BII      ! Galactic Latitude (Radians)
REAL*8 X_EPOC     ! Epoch of RA and DEC (Years)

* INTEGER X_PROJ   ! Projection Section
INTEGER X_PTYTP   ! Projection Type
REAL*8 X_AO       ! Longitude of Projection Center
REAL*8 X_DO       ! Latitude of Center (Radians)
REAL*8 X_PANG     ! Position Angle
INTEGER X_XAXI    ! First Projected axis
INTEGER X_YAXI    ! Second Projected axis

* INTEGER X_SPEC   ! Spectroscopy Section
INTEGER X_ILIN(3) ! Line name X LINE
REAL*8 X_FRES     ! Frequency Resolution (MHz)
REAL*8 X_FIMA     ! Image Frequency (MHz)
REAL*8 X_FREQ     ! Rest Frequency (MHz)
REAL*8 X_VRES     ! Velocity resolution (km/s)
REAL*8 X_VOFF     ! Velocity Offset (km/s)
INTEGER X_FAXI    ! Frequency axis (MHz)

* INTEGER X_RESO   ! Resolution Section
REAL*4 X_MAJ0     ! Major Axis (User Units)
REAL*4 X_MIN0     ! Minor Axis (User Units)
REAL*4 X_POSA     ! Position Angle (Radians)
COMMON /XPAR/X_ITYP,X_FORM,X_NVBI,X_FILL,X_GENE,X_NDIM,X_DIM,
  + X_REF1,X_VAL1,X_INC1,X_REF2,X_VAL2,X_INC2,X_REF3,X_VAL3,
  + X_INC3,X_REF4,X_VAL4,X_INC4,X_BLAN,X_BVAL,X_EVAT,X_EXTR,
  + X_RMIN,X_RMAX,X_MIN1,X_MAX1,X_MIN2,X_MAX2,X_MIN3,X_MAX3,
  + X_MIN4,X_MAX4,X_DESC,X_IUNI,X_ICOD,X_ISYS,X_POSI,X_SOUR,
  + X_RA,X_DEC,X_LII,X_BII,X_EPOC,X_PROJ,X_PTYTP,X_AO,X_DO,
  + X_PANG,X_XAXI,X_YAXI,X_SPEC,X_ILIN,X_FRES,X_FIMA,X_FREQ,
  + X_VRES,X_VOFF,X_FAXI,X_RESO,X_MAJ0,X_MIN0,X_POSA

Basic Header Handling

SUBROUTINE GDFUPIH (IS,NAME,ERROR)
GDF Update Image Header
Update an image header
Arguments :
IS I Image slot Input
NAME C*(*) Gildas Data Frame name Input
ERROR L Logical error flag Output
Subroutines :
-----------------------------------------------

SUBROUTINE GDF_RHSEC(IS, TYPE, ARRAY, SIZE, ERROR)
-----------------------------------------------
GDF Read Header Section
Find information Gildas Data Frame header
Arguments
IS I Image slot number Input
TYPE C*12 Information wanted Input
ARRAY I (*) Array to receive information Output
LENGTH I Size of array Input/Output
ERROR L Logical error flag Output
Subroutines :
SIC_AMBIGS, R4TOR4
-----------------------------------------------

SUBROUTINE GDF_WHSEC(IS, TYPE, ARRAY, SIZE, ERROR)
-----------------------------------------------
GDF Write Header Section
Write a descriptor in Header of file.
Arguments :
IS I Image slot Input
TYPE C*(*) Type of section Input
ARRAY I(*) Address of section Input
LENGTH I Length of section (Words) Input
ERROR L Logical error status Output
Subroutines :
SIC_AMBIGS, R4TOR4
-----------------------------------------------

Named Commons Handling

SUBROUTINE GDF_READX (Islo, Error)
SUBROUTINE GDF_READY (Islo, Error)
SUBROUTINE GDF_READZ (Islo, Error)
-----------------------------------------------
GDF Read Header of the X, Y or Z common area from image slot Islo
ISLO Integer Image Slot number
ERROR Logical Error Flag
-----------------------------------------------

SUBROUTINE GDF_WRITX (Islo, Error)
SUBROUTINE GDF_WRITY (Islo, Error)
SUBROUTINE GDF_WRITZ (Islo, Error)
-----------------------------------------------
GDF Write Header from the X, Y or Z common area to image slot Islo
4 TASK PROGRAMMING MANUAL

<table>
<thead>
<tr>
<th>ISLO</th>
<th>Integer</th>
<th>Image Slot number</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR</td>
<td>Logical</td>
<td>Error Flag</td>
</tr>
</tbody>
</table>

------------------------------------------------------------------------------------------------------------------------

SUBROUTINE GDF_CHXY, GDF_CHYZ, GDF_CHXZ
------------------------------------------------------------------------------------------------------------------------

GDF Copy Header of the X, Y or X common into Y, Z, Z respectively and move memory pointers accordingly.
------------------------------------------------------------------------------------------------------------------------

4.4 Creating GILDAS Tasks

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

- The program, taskname.exe.
- The initialization file, taskname.init, read by commands RUN and SUBMIT to get from the user the task input parameters.
- The checker file, taskname.check, read by commands RUN and SUBMIT, to check the input parameters and pass them to the task.
- A help text file, taskname.hlp

All files must be located in the GILDAS_RUN directory.

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.

4.5 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.

4.5.1 Source code

PROGRAM COMBINE

C------------------------------------------------------------------------------------------------------------------------
C GILDAS Combine in different ways two input images
C (or data cubes)... 
C------------------------------------------------------------------------------------------------------------------------

USE IMAGE_DEF
INCLUDE 'inc:format.inc'
INCLUDE 'inc:errcod.inc'
*
CHARACTER*80 NAMEX,NAMEY,NAMEZ, CODE*20
LOGICAL ERROR
REAL AY,AZ,TY,TZ,B,C
REAL, ALLOCATABLE :: DX(:,,:), DY(:,,:), DZ(:)
INTEGER I, J, N, M, IER, LENC
*
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 = LENC(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
   WRITE(6,* ) ’F-COMBINE, Cannot read input file’
   GOTO 100
ENDIF
*
N = LENC(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
   WRITE(6,* ) ’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%EVAL*1E-7))
   IF (HY%GIL%EVAL.GE.0.0) HY%GIL%EVAL =
   & MAX(HY%GIL%EVAL,ABS(HY%GIL%EVAL*1E-7))
*
* Check input dimensions
DO I=1,4
   IF (HY%GIL%DIM(I).NE.HZ%GIL%DIM(I)) THEN
      N = 1
      DO J=1,4
         N = N*HZ%GIL%DIM(J)
      ENDDO
   ENDIF
   IF (N#.1) THEN
      WRITE(6,* ) ’F-COMBINE, Input images are non coincident’
      GOTO 100
   ELSE
      WRITE(6,* ) ’W-COMBINE, Combining a cube with a plane’
   ENDIF
*
ENDIF
ENDDU
*
CALL GDF_COPY_HEADER(HY,HX)
N = LENC(NAMEX)
IF (N.EQ.0) GOTO 100
CALL SIC_PARSEF(NAMEX(1:N),HXX%FILE,’’,’’.gdf’)
HXX%GIL%BLAN = 2
HXX%GIL%BVAL = B
HXX%GIL%EVAL = 0.0
HXX%GIL%EXTR = 0
*
* 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),DX(N),STAT=IER)
IF (IER.NE.0) THEN
   WRITE(6,*) 'F-COMBINE, Input images are non coincident'
   GOTO 100
ENDF
*
* 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,AZ,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,AZ,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,AZ,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,AZ,TY,
       + HX%GIL%BVAL,C)
   ELSE
      WRITE(6,*) 'Invalid operation code '/CODE
GOTO 100
ENDIF
*
* Write output file
CALL GDF_WRITE_IMAGE(HX,DX,ERROR)
*
STOP 'S-COMBINE, Successful completion'
*
98 WRITE(6,*) 'F-COMBINE, Missing parameter'
100 CALL SYSEXI (FATALE)
END
*
**
SUBROUTINE ADD002(Z,Y,X,N,M,BZ,EZ,TAZ,BY,EY,AY,TY,TX,HX,C)
C-------------------------------------------------------------------------------------------------
C TASK  Internal routine
C  Linear combination of input arrays
C  X = Ay*Y + Az*Z + C
C Arguments
C  Z  R*4(*)  Input array (N)
C  Y  R*4(*)  Input array (N,M)
C  X  R*4(*)  Output array (N,M)
C  N,M  I  Dimensions of arrays
C  BX,BY,BZ  R*4  Blanking values
C  EY,EZ  R*4  Tolerance on blanking
C  AZ,AZ  R*4  Multiplicative factor of array Z, Y
C  TZ,TY  R*4  Threshold on Z,Y
C  C  R*4  Additive constant
C-------------------------------------------------------------------------------------------------
INTEGER N,M
REAL Z(N),Y(N,M),X(N,M),BX,BY,BZ,EY,EZ,AZ,AZ,AY,TY,TX,HX,C
INTEGER I,K
*
DO K=1,M
  DO I=1,N
      X(I,K) = AY*Y(I,K) + AZ*Z(I) + C
    ELSE
      X(I,K) = BX
    ENDIF
  ENDDO
ENDDO
END

4.5.2 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.

4.5.3 Checker File

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.
4.5.4 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).

4.6 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.
5 GTVIRT Programming Manual

5.1 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 Sub-directories 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.

5.2 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
5 GTVIRT PROGRAMMING MANUAL

5.3 Basic Sequence

GreG and the GTVIRT are intimately related, and it is not recommended to use the GTVIRT without GreG. Accordingly, 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 ... (‘Append’) ! 3
CALL GTVIEW (‘Append’) ! 4

1. Open a new graphic segment. This routine is an “intelligent” routine checking whether pen attributes have been changed, and calling both GTSEG (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’)

5.4 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.
5.5 Subroutines

- gtalp
  Commute to alphanumeric mode. On Windows system, transfer focus to the terminal window.

- gtchar
  Draws a character string.

- gtclr
  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

- gtcpl
  Raise alphanumeric window, and returns focus to it.

- gtcurv
  Call the cursor

- gtcls
  Delete the last graphic segment

- gtdraw
  Draw a vector from current pen position to the specified point.

- gtedit
  Edit the current segment properties.

- gterflag, gtergto, gtertst
  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

- gtpoly
  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 GTVRT mode (Sleep or Wake up), and returns previous mode.
- `gtx_newimage`
  Create a new image slot
- `gtx_image`
  Draw a bitmap to an image slot
- `gtx_numimage`
  Get a free image slot
- `gtx_majimage`
  Define parameters of an image slot
- `gtx_fillpoly`
  Fill a closed polygon.
- `init_gtvirt`
  Initialize the GTVRT 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
- `gtx beep`: beep
- `gtx clear`: clear the current window
- `gtx draw`: draw line to current
- `gtx flush`: flush the normal drawing buffer.
- `gtx pen`: select the immediate pen
- `gtx poly`: draw a polyline
- `gtx reloc`: relocate the immediate pen
- `gtx where`: returns immediate pen position
Index

EXAMPLE
   ENTER_USER, 7
   EXEC_USER, 7
   PLAY_USER, 7
   RUN_USER, 8
   US_ERROR, 11
   US_EXEC, 11

ROUTINES
   FREE_VM, 22
   GAGOUT, 13
   LENC, 22
   SIC_AMBIGS, 20
   SIC_ANALYSE, 12, 21
   SIC_CH, 10, 19
   SIC_CTRL, 19
   SIC_DEF_CHAR, 14
   SIC_DEF_DBLE, 14
   SIC_DEF_FUNC, 16
   SIC_DEF_INTB, 14
   SIC_DEF_LOGH, 14
   SIC_DEF_REAL, 14
   SIC_DEL SYMBOL, 22
   SIC_DEL_VARIABLE, 16
   SIC_DESCRIPTOR, 15
   SIC_FORMAT, 12, 21
   SIC_GET_CHAR, 14
   SIC_GET_DBLE, 14
   SIC_GET_INTB, 14
   SIC_GET_LOGH, 14
   SIC_GET_REAL, 14
   SIC_GETINV, 13
   SIC_GET_SYMBOL, 22
   SIC_GETVM, 21
   SIC_H, 10, 19
   SIC_INCARNE, 16
   SIC_INSERT, 18
   SIC_INTER_STATE, 19
   SIC_K, 10, 19
   SIC_L, 10, 19
   SIC_LANG, 12, 19
   SIC_LEN, 21
   SIC_LET_CHAR, 14
   SIC_LET_DBLE, 14
   SIC_LET_INTB, 14
   SIC_LET_LOGH, 14
   SIC_LET_REAL, 14
   SIC_LIRE, 18, 19
   SIC_LOAD, 4, 12, 18
   SIC_LOG, 19
   SIC_LOGICAL, 16

   SIC_LOWER, 22
   SIC_MATH, 16
   SIC_NARG, 20
   SIC_NEXT, 21
   SIC_OPT, 5, 18
   SIC_PARSE, 22
   SIC_PRESENT, 20
   SIC_R, 10, 19
   SIC_R8, 10, 19
   SIC_RUN, 7, 17
   SIC_SET_SYMBOL, 22
   SIC_SEX, 16
   SIC_START, 21
   SIC_UPPER, 22
   SIC_VOLATILE, 16
   SIC_WPR, 22
   SIC_WPRN, 22
   VAR_TYPE, 15

   SIC_MEMORY, 5