CERN Program Library Long Writeup I102

KUIP

Kit for a User Interface Package

Version 2.05

Application Software Group

Computers and Network Division

CERN Geneva, Switzerland

```
CERN Program Library Office
CERN-CN Division
CH-1211 Geneva 23
Switzerland
Tel.  +41 22 767 4951
Fax.  +41 22 767 7155
Bitnet:  CERNLIB@CERNVM
DECnet:  VXCERN::CERNLIB (node 22.190)
Internet:  CERNLIB@CERNVM.CERN.CH
```


Contact Person: Nicole Cremel /CN (N.Cremel@cern.ch)

Technical Realization: Alfred Nathaniel /CN (A.Nathaniel@cern.ch)

Edition – October 1994

Acknowledgments

Many people participated in the design and the implementation of KUIP. The first version of KUIP released in 1987 was designed and implemented by René Brun and Pietro Zanarini. Many basic features stem from the package ZCEDEX[1], implemented at CERN in 1982 by R. Brun, C. Kersters, D. Moffat and A. Petrilli, which offered already command parsing, macros, vectors, and functions.

The development of KUIP was started in the context of PAW, the *Physics Analysis Workstation* project, and therefore everybody in the PAW team must be acknowledged. Olivier Couet implemented the graphics menus and the graphics interface of KUIP to HIGZ. Achille Petrilli, Fons Rademakers, and Federico Carminati wrote the original routines for break interception on various platforms. Carlo Vandoni integrated the functionality of SIGMA[2] (*System for Interactive Mathematical Applications*). Colin Caughie (Edinburgh) implemented new features in macro flow control. Ilias Goulas (Turin) provided KUIB (KUIP Interface Builder) as a replacement for the original KUIPC (KUIP Compiler). Alain Michalon (Strasbourg) and Harald Butenschoen (DESY) ported KUIP to the MVS/TSO and NEWLIB environments. Valeri Fine (Dubna) ported KUIP to MSDOS and Windows/NT. C.W. Hobbs (DEC) provided the terminal communication routines for the VMS version of KUIP/Motif.

The maintenance of the overall package and developments for the basic part and KUIPC is in the hands of Alfred Nathaniel. Nicole Cremel is responsible for the KUIP interface to OSF/Motif. Fons Rademakers was in charge of the overall maintenance before that and made many important contributions to the KUIP/Motif interface.

About this manual

Many features described in this manual edition are available only since the `94b` release of the CERN libraries. For example, there were many enhancements added to the macro interpreter.

The coding of some features (e.g. global variables) did not meet the deadline for the `94b` release. They are already documented here with the remark that they are implemented only `95a` pre-release in the `new` area.

Therefore, if you find something described in this manual does not work in your program, please check that you are using an up-to-date version of KUIP.

The manual is structured in the following way:

- Chapter 1 gives a short overview of what KUIP is doing.
- Chapter 2 is intended for all application *users* describing the user interface provided by KUIP.

The remaining parts of the manual are intended for application *writers*:

- Chapter 3 describes in the first part how to define the commands to be handled by the application. The second part explains how to use the features provided by KUIP/Motif.
- Chapter 4 is a reference for the calling sequences of KUIP routines.
- Chapter 5 is a reference for KUIP built-in commands.
- Appendix A gives an example for a simple KUIP-based application program.
- Appendix B gives some useful information about the system dependencies in interfacing Fortran and C.

Throughout this manual we use `mono-type face` for examples. Since KUIP is mostly case-insensitive, we use `UPPERCASE` for keywords while the illustrative parts are written in `lowercase`. When mixing program output and user-typed commands the user input is `underlined`.

In the index the page where a command or routine is defined is in **bold**, page numbers where they are referenced are in normal type.

This document was produced using LaTeX [3] with the `cernman` style option, developed at CERN. A PostScript file `kuip.ps`, containing a printable version of this manual, can be obtained by anonymous ftp as follows (commands to be typed by the user are underlined):

```
ftp asis01.cern.ch
Connected to asis01.cern.ch.
220 asis01 FTP server (...) ready
Name (asis01.cern.ch:your-name): ftp
Password: your-name@your-host
ftp> cd cernlib/doc/ps.dir
ftp> bin
ftp> get kuip.ps
ftp> quit
```

Note that a printer with a fair amount of memory is needed in order to print this manual.

There is a Usenet news group `cern.heplib` being the forum for discussions about KUIP and the other packages of the CERN program library. Bug reports, requests for new features, and questions of general interest should be sent there. People without Usenet access can subscribe to a distribution list by sending a mail to `listserv@cernvm.cern.ch` containing the message

```
SUBSCRIBE HEPLIB
```

Table of Contents

List of Tables

List of Figures

1 User Interface Management Systems and KUIP

KUIP (Kit for a User Interface Package) is the User Interface system developed at CERN in the context of PAW, the Physics Analysis Workstation system [4, 5].

A User Interface Management System (UIMS) is a software toolkit intended to:

– provide a homogeneous environment in which different kinds of **users** interact with different kinds of applications (see Figure 1.1).
– provide tools to help the programmer in developing an interactive application.

Each application usually has a heterogeneous user base at different levels of experience. The design of a UIMS should aim at a good compromise between the ease of use for beginners and the avoidance of frustration for more experienced users. A beginner or casual user may prefer a menu mode for guiding him through the set of command, while a user who is already familiar with an application can often work more efficiently with a command line mode.

Both requirements can only be met by a **multi-modal dialogue** system, i.e. the application has to provide different dialogue styles with the possibility to switch between them from inside the application.

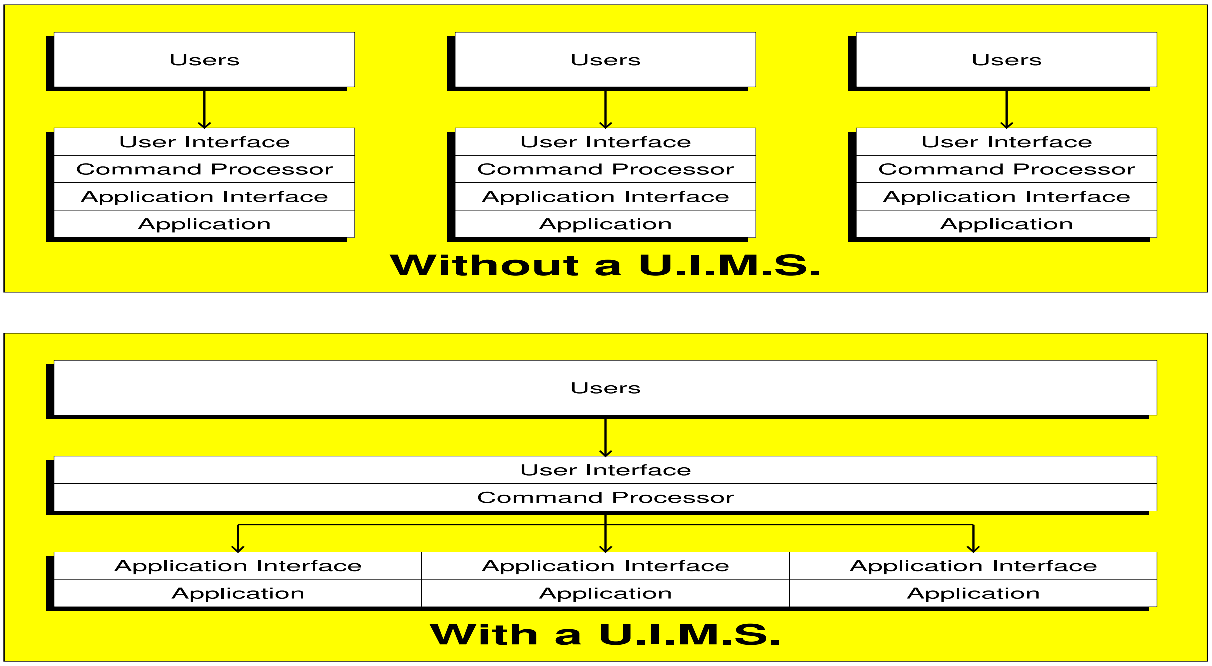


Figure 1.1: The homogeneous environment provided by a U.I.M.S.

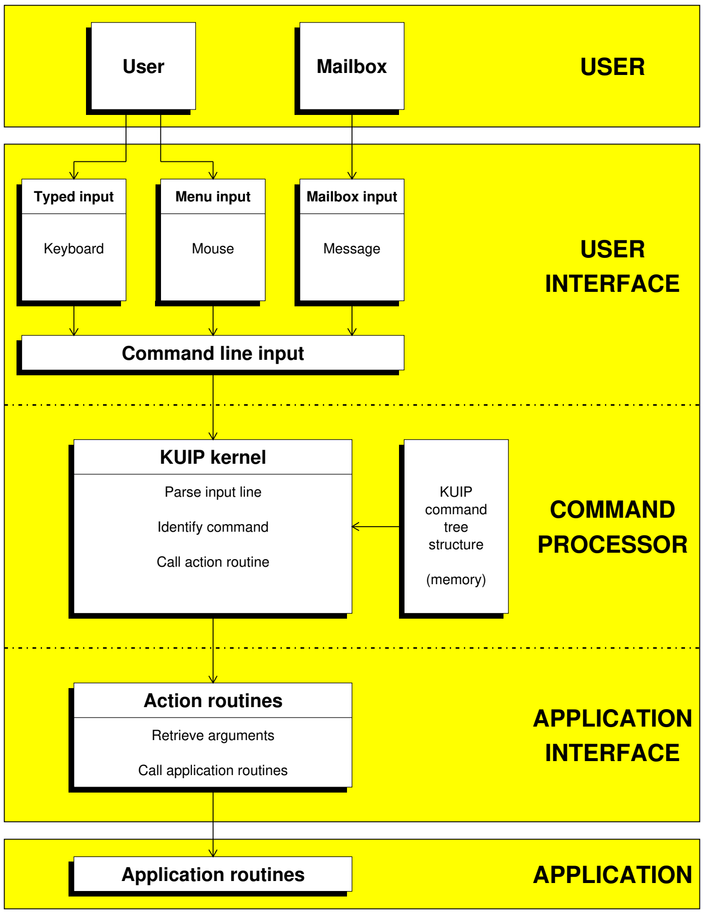


Figure 1.2: The different layers in a KUIP application

In any case the UIMS should allow to include enough **online-help** to make the application usable without any additional written documentation.

Another important point is to allow **mixed control** of command execution. In the usual case the command processor prompts the user for the next command and passes it onto the application. On the other hand the application should also be able to pass command sequences back to the command processor for execution.

1.1 The Layers of KUIP

As a User Interface system KUIP concerns both the application writer and the application user. Figure 1.2 shows the different layers in a KUIP-based application. The ordinary user only has to know about the top layer with standard rules how to select a command and how to supply the necessary arguments.

The application writer on the other hand has to understand the tools which allow to define the command structure for the middle layer. He also has to provide the bottom layer which implements the specific code for each command.

1.1.1 The Application Writer's View

The application writer has to describe the command tree structure and the parameters and action routines associated with each command. In order to do its job KUIP has to store this information in internal structures. The possibility that the application programmer has to write himself the routine calling the appropriate KUIP routines was considered as being too inconvenient.

Instead the application writer only has to provide a text file called the Command Definition File (CDF) containing a formal description of the command set. A stand-alone program, the KUIP Compiler (KUIPC), analyzes the CDF and generates a file containing the source code (i.e. calls to KUIP routines) to store the command structure at run-time.

The routine generated by KUIPC has to be called by the application program once between the initialization of KUIP (`KUINIT`) and the point when control is passed to the KUIP main input loop (`KUWHAT`). For the commands given by the user KUIP calls the associated action routines which have to be provided by the application writer. The action routine can then retrieve the command arguments (`KUGETx`) and perform the appropriate actions. The CDF allows to specify parameters as being mandatory (for which the user must supply an argument value) or optional (for which the `KUGETx` routines return a default value if omitted by the user).

Generating the actual command definition code automatically from the higher-level CDF format offers many advantages:

– The directives needed to mark-up the CDF description are easier to learn than the calling sequences and correct ordering of the definition routines.
– The command structure is better visible in the CDF than in the corresponding source code cluttered by calls to cryptic routine names with cryptic option arguments.
– The CDF is far easier to edit because the writer does not have to worry about continuation lines or the correct quoting of character strings.
– KUIPC gives the choice between generating C or Fortran source code. Using the C output mode can considerably reduce an application's start-up time. KUIPC can allocate most of the structures statically that building the command tree involves only a few pointer manipulations. On the other hand the Fortran output mode allows to keep the installation procedures simple for otherwise purely Fortran-based applications.

1.1.2 The Application User's View

KUIP provides different dialogue modes (or styles) how the user can enter commands: the default command line input from the keyboard and various menu modes either driven by keyboard input or by mouse clicks. Switching between dialogue styles is possible at any moment during the interactive session. This makes KUIP suitable to applications with a heterogeneous user base: each user can choose according to his taste and knowledge of the application.

In command line mode KUIP writes out a prompt and waits for input from the user. The input consists of a command name possibly followed by an argument list. The command name can be abbreviated and KUIP matches it against the set of defined commands. Then the arguments given on the input line are assigned to the command parameters. If any of the mandatory arguments is missing the user is prompted to enter a value for them. If a character string appears in a position where a number is expected or if a numeric value is outside the allowed range the user is prompted again to correct the argument. Only when the input passes these basic consistency checks KUIP calls the action routine and issues the next command line prompt.

1.2 A Quick Look at the Main Features of KUIP

KUIP represents a general-purpose User Interface Management Systems. The basis of KUIP is the so-called Command Definition File (CDF) containing a description of the command parameters, on-line help information, and how the commands are grouped into menus. Since menus can be linked to other menus the application commands are represented by an inverted tree in analogy to a Unix file system. The KUIP Compiler (KUIPC) converts the CDF into routines which have to be compiled and linked with the interactive application.

The interaction between the user and the application is either by typing command lines or selecting command from alphanumeric or graphical menus. The user is able to switch between dialogue modes at any time. In menu mode the user can traverse the command tree and select commands for execution. This does not require any additional programming from the application writer since the menu structure is already described in the CDF.

The commands typed-in may be abbreviated by omitting parts of the complete command path as long as this does not produce ambiguities between different commands. Previous command lines can be recalled, edited, and re-executed. A `csh`-like history mechanism is also available.

KUIP provides a macro language with variables, expressions, and various control flow constructs which allows to execute a complex sequence of commands by typing a single `EXEC` command. An application can execute a logon macro at start-up time that the user can configure the environment to his taste. All command lines entered during a sessions are recorded in form of a macro file which can be the starting point for a proper macro. An application can also be run in batch mode by executing a macro file without any user interaction.

The documentation for each command is contained inside the CDF. Keeping the CDF up-to-date guarantees that the on-line help derived from it is always in phase with the actual program version. KUIP allows to write out the command description marked-up with LaTeX formatting command which then can be included in the proper users' guide or reference manual.

1.3 The Advantages of Using KUIP/Motif

Motif [6] is a widget set developed by the Open Software Foundation (OSF). Most major computer vendors joined OSF and support Motif as part of their system software.

KUIP/Motif is an extension to the basic KUIP package which interfaces to the Motif windowing system. Again, the development of KUIP/Motif started off in the context of PAW. The aim was to generalize the ideas which went into the Motif version PAW++ and make them available to other, already existing KUIP-based applications.

As a result an application programmer can supply his users a powerful windowing interface with minimum effort. By merely changing a few KUIP initialization calls the application inherits already most of the KUIP/Motif functionalities (see figures 2.3 and 2.4):

- A terminal window allows to type commands and to scroll through the application output messages.
- A general object browser visualizes the command tree and file system structure. The browser window can be "cloned" to look at different parts of the object trees at the same time. Commands can be invoked by browsing through the command tree or from pull-down menus attached to the browser window. Arguments can be filled into command panels showing the completely list of command parameters and option values.
- User defined panels allow to execute command sequences by a single mouse click.
- KUIP/Motif cooperates with HIGZ/X11 and allows for several simultaneous graphics windows.

In order to take full advantage of the KUIP/Motif facilities the application writer has to spend only a little more effort. The central point of KUIP/Motif is the object browser. Every application deals with some kind of "objects" which are often linked into a hierarchical tree structure. The KUIP/Motif browser allows the user to traverse and visualize the content of the tree and to operate on individual objects.

The actual nature of these "objects" is arbitrary. For example, in PAW the prime objects are histograms and N-tuples, while in GEANT they are the volumes in the detector geometry or the particle tracks in an event simulation. Application-specific objects are integrated into the browser defining the possible objects types in the CDF. The only additional coding work required is to provide a routine which, given the path selected in the browser window, scans the directory content and returns for each object its identification (section 3.3).

An object is identified by its name and its type. The type names or "classes" are defined in the CDF and determine the icon used for showing the object and the list of possible actions to operate on a selected object. The value returned as object "name" is up to the scanning routine but naturally it should be the same usually required to refer to the object in a command. Behind the class-specific action menus there are command sequences which allow to insert the object name in the appropriate position. If the object name is the only item necessary to make the command complete it can be executed immediately, otherwise the command panel pops up where the user can enter the missing arguments.

One of the main advantages of KUIP/Motif is that all applications will give the users the same "look and fill". Further design goals met by KUIP/Motif are:

- KUIP/Motif can be used without any prior knowledge about Motif programming. On the other hand KUIP/Motif provides the hooks to integrate application specific Motif widgets into the user interface.
- For an existing KUIP based application a Motif interface can be provided by changing of a few initialization calls only.
- In order to exploit to the full power of KUIP/Motif the application writer has to add new routines rather than to change existing ones. Therefore the whole application code can reside in a single library, while the different initialization calls between basic KUIP and KUIP/Motif can be absorbed in the main program.
- For convenience a single module can contain both the basic KUIP command line interface and the KUIP/Motif interface giving the user the choice at startup time. Loading the Motif libraries adds typically 2–3 Mbytes to the module size. If memory is at a premium a module with the basic KUIP command line can be generated which does not required any Motif specific code to be loaded.
- Although KUIP/Motif is fully integrated with the HIGZ/X11 graphics package a non-graphical application does not need to load any HIGZ code.

1.4 Implementation and Portability

Originally KUIP was written at CERN completely in Fortran 77. The choice of Fortran as implementation language was governed by the fact that at that time (1987) the Fortran compiler was the only one commonly available on all initial target platforms (VM/CMS, VAX/VMS, and Apollo). However, Fortran misses a number of language features which are essential for programming a User Interface package: recursivity, function pointers, and recovery from exceptions.

Already with the advent of Unix workstations some system dependent tasks could not be expressed in Fortran anymore and had to be written in C. The use of KUIP in various applications with widely different requirements made evident another limitation of Fortran: The purely static allocation of character variables leads to a trade-off between wasted memory space and the risk that one application could still need more than the fixed size limit.

For the KUIP version released in the beginning of 1992 major parts were rewritten in C. The rewrite removed most of the size limitations, added new functionalities, and at the same time improved the portability by placing non-standard Fortran with standard C constructs. Storing the command tree in C structures also simplified the implementation of the Motif interface (which was written in C from the very beginning) considerably because it obsoleted routines previously needed for accessing the information stored inside ZEBRA structures.

Two main parts of KUIP are still mainly in Fortran: the handling of vectors and the macro compiler/interpreter. Both have a number of limitations which can be avoided using C. The intention is to replace them by C code as well, and at the same time formalize the interface for using KUIP directly from applications written in C.

KUIP is part of the CERN PACKLIB and partially depends on other standard packages included in this library. Several large CERN application programs use KUIP: PAW[4], GEANT[7], and CMZ[8] amongst others. The basic KUIP has also been ported to the following vendor/operation systems where underlining indicates the platforms for which ready-to-use libraries are available from the CERN program library:

Convex: Convex-OS
Cray: Unicos (`cft77` compiler for X/MP)
DEC: Vax/VMS, RISC/Ultrix, Vax/Ultrix, Alpha/VMS, Alpha/OSF, Alpha/Windows-NT
HP: Apollo/Domain-OS, HP/UX
IBM: VM/CMS, MVS/TSO, NEWLIB (DESY MVS), RS-6000/AIX
SGI: Irix
Sun: Sun-OS, Solaris
PC: MS-DOS (f2c), NeXT (Absoft-Fortran), Linux (f2c)

KUIP/Motif requires the Motif libraries version 1.1 or later. It is known to be working on the following platforms:

DEC: RISC/Ultrix V4.3, Vax/VMS, Alpha/VMS, Alpha/OSF
HP: Apollo/Domain-OS SR10.4 (not for DN10000), HP/UX
IBM: RS-6000/AIX
SGI: Irix
Sun: Sun-OS (with DEC-Motif libraries and using the DEC-Motif window manager), Solaris

2 User interface

2.1 Dialogue Styles

In a KUIP-based application the user has the choice of different dialogue styles ranging from the conventional command line interface to a high-level windowed environment based on OSF/Motif. The optimum choice for each user depends on the level of experience and also on the type of terminal or workstation available. The user can freely switch between the different styles from within the application using the command `STYLE`. The only exception is the Motif interface since its internal structure is completely different from the others.

Underneath of all these different dialogue style lies a common processing engine. The command and its arguments is extracted from the input which was either typed in directly by the user or constructed by one of the menu styles and then the appropriate application routine is called. Even when using a menu-driven style a basic knowledge of the command processing is necessary, e.g. how commands can be abbreviated and what is the meaning of special characters. Before plunging into the details of the command syntax we want to overview the capabilities of the individual styles.

2.1.1 Alphanumeric modes

The styles “`C`”, “`AN`”, and “`AL`” can be used with any kind of terminal device.

2.1.1.1 STYLE C

`STYLE C` is the basic keyboard input mode: the user types a command line which is then executed. After being more familiar with the command set of an application, most users actually prefer `STYLE C` to type in everything to avoid the frequent switching between mouse and keyboard input required by graphical menu techniques.

KUIP provides an integrated online help facility. The first line to type in an unfamiliar application is “`HELP`” (or “`help`” — KUIP command names are case-insensitive) which shows the top level of the command tree. Taking PAW as one of the prime examples of a KUIP-based application we get the following output:

```
PAW > help

From /...

 1:   KUIP          Command Processor commands.
 2:   MACRO         Macro Processor commands.
 3:   VECTOR        Vector Processor commands.
 4:   HISTOGRAM     Manipulation of histograms, Ntuples.
```

```
 5:   FUNCTION      Operations with Functions. Creation and plotting.
 6:   NTUPLE        Ntuple creation and related operations.
 7:   GRAPHICS      Interface to the graphics packages HPLOT and HIGZ.
 8:   PICTURE       Creation and manipulation of HIGZ pictures.
 9:   ZEBRA         Interfaces to the ZEBRA RZ, FZ and DZ packages.
10:   FORTRAN       Interface to MINUIT, COMIS, SIGMA and FORTRAN Input/Output.
11:   NETWORK       To access files on remote computers.
12:   OBSOLETE      Obsolete commands.

Enter a number ('Q'=command mode):
```

At this point we can either enter "Q" to return to the command line prompt again, or we can investigate the command tree. Typing "1" show us the commands and sub-menus which are linked to the menu "/KUIP":

```
Enter a number ('Q'=command mode): 1

   /KUIP

   Command Processor commands.

From  /KUIP/...

 1: * HELP          Give the help of a command.
 2: * USAGE         Give the syntax of a command.
 3: * MANUAL        Write on a file the text formatted help of a command.
 4: * EDIT          Invoke the editor on the file.
 5: * PRINT         Send a file to the printer.
 6: * PSVIEW        Invoke the PostScript viewer on the file.
 7: * LAST          Perform various operations with the history file.
 8: * MESSAGE       Write a message string on the terminal.
 9: * SHELL         Execute a command of the host operating system.
10: * WAIT          Make a pause (e.g. inside a macro).
11: * IDLE          Execute a command if program is idle.
12: * UNITS         List all Input/Output logical units currently open.
13: * EXIT          End of the interactive session.
14: * QUIT          End of the interactive session.
15:   FUNCTIONS            *** KUIP System Functions ***
16:   ALIAS         Operations with aliases.
17:   SET_SHOW      Set or show various KUIP parameters and options.

Enter a number ('\'=one level back, 'Q'=command mode):
```

The menus /KUIP and /MACRO contain the KUIP built-in commands which are common to all applications. Now we could continue wandering around the command tree by choosing one of the sub-menus or by entering "\" to go back to the previous menu. Instead we can also select one of the actual commands which are marked by "*":

```
Enter a number ('\'=one level back, 'Q'=command mode): 1

 * KUIP/HELP [ ITEM OPTION ]
```

```
    ITEM       C 'Command or menu path' D=' '
    OPTION     C 'View mode' D='N'

    Possible OPTION values are:

     EDIT    The help text is written to a file and the editor is invoked,
     E       Same as 'EDIT'.
     NOEDIT  The help text is output on the terminal output.
     N       Same as 'NOEDIT'

    Give the help of a command.  If ITEM is a command its full explanation is
    given:  syntax (as given by the command USAGE), functionality, list of
    parameters with their attributes (prompt, type, default, range, etc.).  If
    ITEM='/' the help for all commands is given.

    If HELP is entered without parameters or ITEM is a sub-menu, the dialogue
    style is switched to 'AN', guiding the user in traversing the tree command
    structure.

    'HELP -EDIT' (or just 'HELP -E') switches to edit mode:  instead of writing
    the help text to the terminal output, it is written into a temporary file
    and the pager or editor defined by the command HOST_PAGER is invoked.  (On
    Unix workstations the pager can be defined to display the help text
    asynchronously in a separated window.) 'HELP -NOEDIT' (or just 'HELP -N')
    switches back to standard mode.  The startup value is system dependent.

 <CR>=continue, 'Q'=command mode, 'X'=execute:
```

Here we see the help text for the `HELP` command itself. In fact the command is not called "`HELP`" but "`/KUIP/HELP`" and has two parameters. Nevertheless KUIP recognized the abbreviation and provided default arguments ("`D=...`") which allowed the command to be executed.

Typing "`X`" allows to execute the current command — for the `HELP` command itself this would not be very useful. Simply hitting the RETURN or ENTER-key would brings us back to the previous menu. Instead we want to use "`Q`" which brings us back to the command line mode.

As we have seen the `HELP` command actually can take two arguments. The second one we ignore for the moment since the first one is much more important. If we know the name of a command but do not know what it is doing exactly we can get the help text directly without going through the help menu mode:

```
 PAW > help usage

  * KUIP/USAGE ITEM

    ITEM       C 'Command name'

    Give the syntax of a command.  If ITEM='/' the syntax of all commands is
    given.
```

The command USAGE is also part of the online help system. It shows the first line only of the help information in case we know what a command is supposed to do but are not sure about the number and order or arguments:

```
PAW > usage manual

 * KUIP/MANUAL ITEM [ OUTPUT OPTION ]
```

This tells us that the MANUAL command takes three arguments of which the last two are optional. Although we do not know yet what this command is doing we try it nevertheless:

```
PAW > manual
Command or menu path (<CR>=manual)

 * KUIP/MANUAL ITEM [ OUTPUT OPTION ]

   ITEM       C 'Command or menu path'
   OUTPUT     C 'Output file name' D=' '
   OPTION     C 'Text formatting system' D=' '

   Possible OPTION values are:

   ' '     plain text : plain text format
    LATEX  LaTeX format (encapsulated)
    TEX    LaTeX format (without header)

   Write on a file the text formatted help of a command.  If ITEM is a menu
   path the help for all commands linked to that menu is written.  If ITEM='/'
   the help for the complete command tree is written.  If OUTPUT=' ' the text
   is written to the terminal.

   The output file produced with option LATEX can be processed directly by
   LaTeX, i.e. it contains a standard header defining the meta commands used
   for formatting the document body.  With option TEX only the document body
   is written into the output file which can be included by a driver file
   containing customized definitions of the standard meta commands.  Example:

    MANUAL / MAN.TEX LATEX

   will produce the file MAN.TEX containing the documentation of all available
   commands in LaTeX format.
```

KUIP realized that we did not give any argument. Since the first parameter ITEM is mandatory the command cannot be executed without one. We were prompted for it and accepted the proposed default value. (It is a peculiarity of the MANUAL command that the default depends on the preceding HELP, USAGE, or MANUAL command. Usually it is either a value fixed by the application writer or the one used in the previous execution of the very same command.)

The output of this MANUAL command looks exactly the same as for the corresponding HELP command. The behavior becomes different, however, when ITEM is not a command but a menu name. "HELP KUIP" enters the help menu mode starting off at the /KUIP menu instead of the root menu "/". "MANUAL KUIP" on the other hand shows the help text for all commands linked to this menu and all of its sub-menus. Furthermore the output can be redirected into a file and beefed up with LaTeX formatting directives.

2.1.1.2 STYLE AN

`STYLE AN` is very similar to the help menu mode in `STYLE C`. The only difference is that when selecting a command the question

```
<CR>=continue, 'Q'=command mode, 'X'=execute:
```

is never asked.

2.1.1.3 STYLE AL

`STYLE AL` is a slight variation of `STYLE AN` using letters instead of numbers to label the items:

```
PAW > style al

From /...

 A:   KUIP          Command Processor commands.
 B:   MACRO         Macro Processor commands.
 C:   VECTOR        Vector Processor commands.
 D:   HISTOGRAM     Manipulation of histograms, Ntuples.
 E:   FUNCTION      Operations with Functions. Creation and plotting.
 F:   NTUPLE        Ntuple creation and related operations.
 G:   GRAPHICS      Interface to the graphics packages HPLOT and HIGZ.
 H:   PICTURE       Creation and manipulation of HIGZ pictures.
 I:   ZEBRA         Interfaces to the ZEBRA RZ, FZ and DZ packages.
 J:   FORTRAN       Interface to MINUIT, COMIS, SIGMA and FORTRAN Input/Output.
 K:   NETWORK       To access files on remote computers.
 L:   OBSOLETE      Obsolete commands.

Enter a letter ('Q'=command mode):
```

2.1.2 Graphics modes

The graphics styles `G` and `GP` can only be used on workstations. Furthermore they are only operational if the application writer has chosen to do so. (The KUIP main loop has to be invoked by calling `KUWHAG` instead of `KUWHAT`. This implies to link the application with the graphics library (e.g. `GRAFX11` or `GRAFGPR`) containing the appropriate HIGZ version for the underlying windowing system.

The user interacts with the application by moving the mouse pointer to appropriate box and clicking with the left mouse button. If the corresponding action requires additional keyboard input the mouse pointer changes into a question mark shape. Pressing the right mouse button leaves the graphics mode and reverts to style `C`.

The style names `G` and `GP` allow additional attributes `S` and `W` which may also be combined, e.g. `GSW`. The attribute `S` uses software characters for writing texts in case the hardware font does not have the right size. The attribute `W` uses a shadow width effect to give a 3D impression.

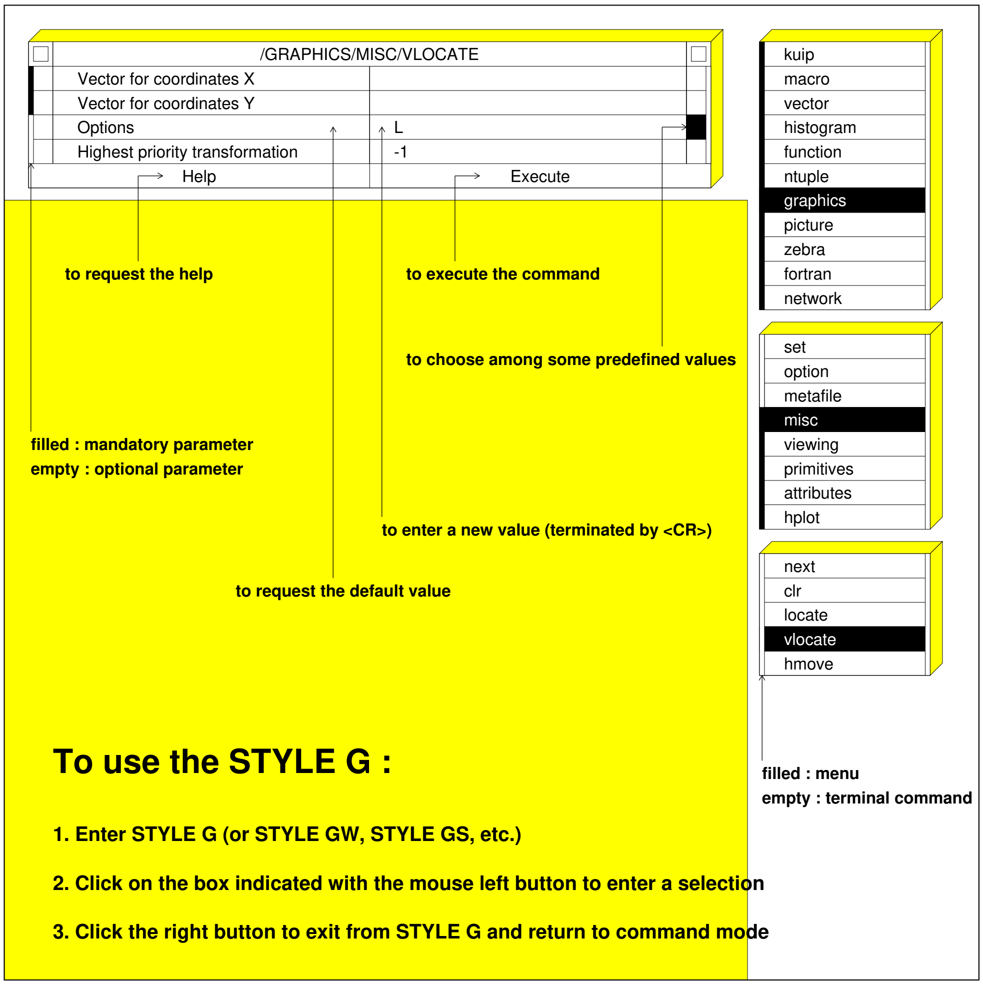


Figure 2.1: Example of STYLE G

2.1.2.1 STYLE G

`STYLE G` is similar to `STYLE AN` but the menus are displayed in the graphics area (see figure 2.1). This restricts the part of the graphics window available for other graphics output to the shaded area. The actual size of the reserved area can be adjusted by additional arguments to the `STYLE` command.

Menus are marked by black vertical bars and the presently active menu levels are high-lighted. When finally a command name is selected the panel displayed in the top left corner shows the parameter descriptions and allows to set argument values and execute the command.

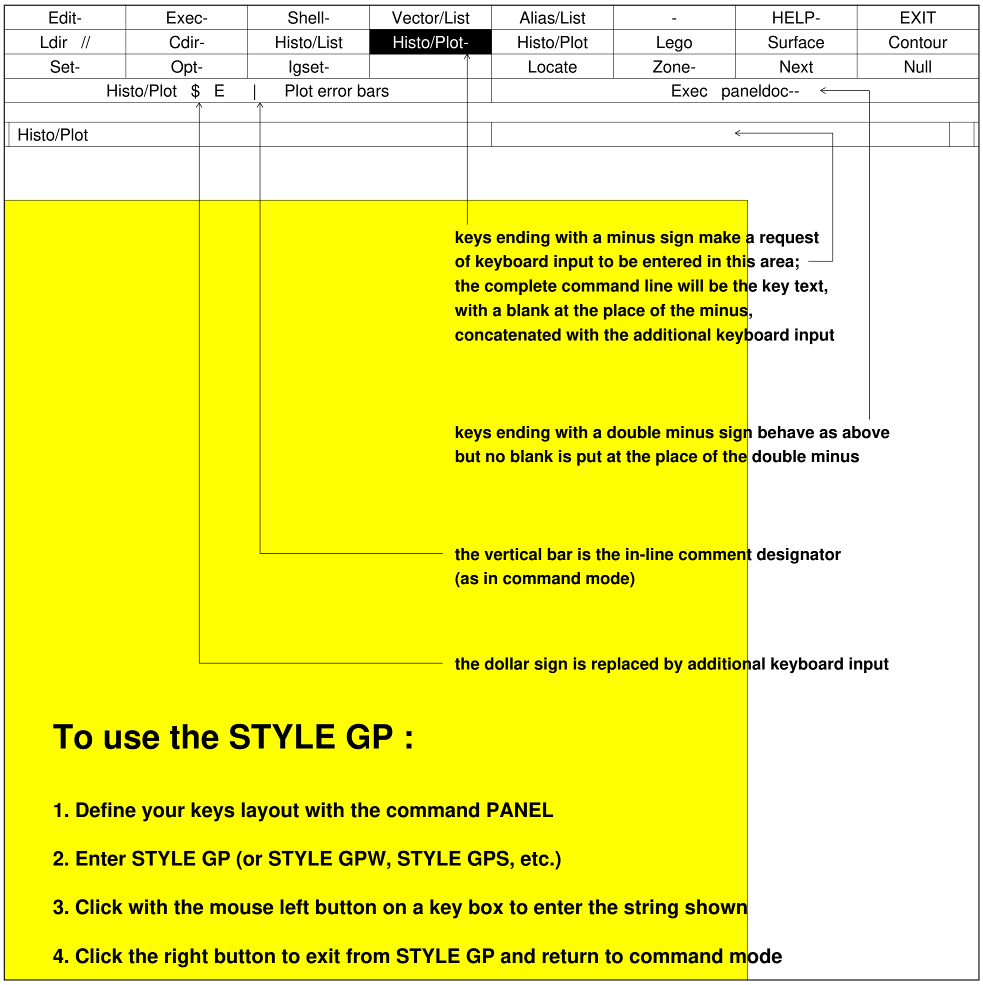


Figure 2.2: Example of STYLE GP

2.1.2.2 STYLE GP

`STYLE GP` displays a user-defined panel in the top part of the graphics area (see figure 2.2). The layout and content of the panel is defined by the `PANEL` command. (Actually, if a panel is defined it is also displayed by style `G`. Therefore style `GP` has to be read as "panel-only" rather than "panel" style.)

The content of each panel box gives the command which is executed when clicked on it. Some characters have a special meaning in the panel command sequences:

```
panel 0
panel 1 Edit-     Exec- Shell-     Vector/List Alias/List -     HELP-   EXIT
panel 2 'Ldir //' Cdir- Histo/List Histo/Plot- Histo/Plot Lego  Surface Contour
panel 3 Set-      Opt-  Igset-     ' '         Locate     Zone- Next    Null
panel 4 'Histo/Plot $ E | Plot error bars'     'Exec paneldoc--'
```

Everything following "`|`" is ignored. "`$`" requests and is replaced by keyboard input. A single "`-`" at the end of the command sequence is replaced by the keyboard input separated by a blank. (This minimizes the need for quotes in the PANEL command, e.g. "`Edit-`" instead of "`'Edit $'`".) The blank separation is suppressed if the command ends with "`--`".

Style GP is very useful for applications where a limited set of commands has to be executed frequently, for example in an online monitoring package or in an event display program. The panel definition and the switch to style GP could actually be contained in file which is executed at startup time. Whilst being in style GP the panel can also be redefined. For example, clicking on "`Exec paneldoc--`" and entering "`1`" executes the macro file `paneldoc1.kumac` which could contain:

```
panel 0
panel 1 'Histo/Plot 1' 'Histo/Plot 2' 'Histo/Plot 3' 'Histo/Plot 4'
panel 2 'Histo/Plot 5' 'Histo/Plot 6' 'Histo/Plot 7' 'Histo/Plot 8'
panel 3 'Cdir //LUN1' 'Cdir //LUN2'
panel 4 'Exec paneldoc | to go back'
```

2.1.2.3 STYLE XM

MOTIF is not a "KUIP STYLE" in the same sense as "STYLE G" or "STYLE GP". When an application has decided to use KUIP/Motif switching to another style is not possible. This is not a real drawback in the sense that all the features provided by the other styles are included in KUIP/Motif (e.g. command mode and panel(s) handling).

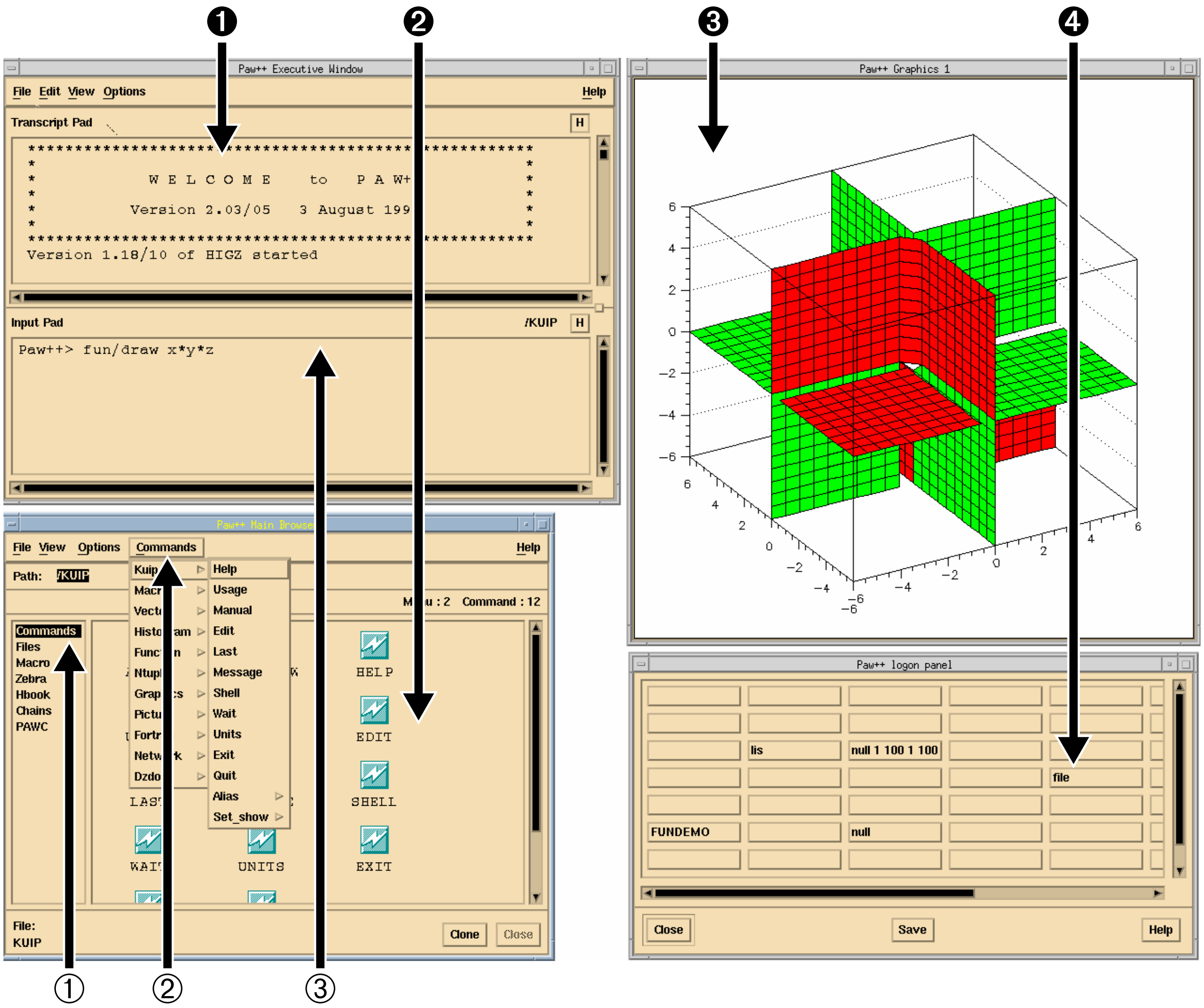
An application based on KUIP/Motif will always start two processes, which will give at least two windows (Fig. 2.3):

– The **"Executive Window"** ❶ is a terminal emulator built in inside KUIP. It enables a powerful "command input and output" window that experienced users can use to type full commands with list of parameters. More information will be given on the main features of this terminal emulator in the part 2.8.2.
– The **"Main Browser"** ❷ window is a general tool to display and traverse a hierarchical directory structure of objects which are defined either by KUIP itself (commands, files, macros) or by the application (e.g. in PAW++: Zebra and Hbook files, Chains, ...). This is in many ways similar to the well-known browsers in the PC/MAC utilities or the visual tools on some workstations. For more information on the browser interface see section 2.8.1.

Many other windows may exist or be created later on depending on the application:

– Optional HIGZ **Graphics Window**(s) ❸ with object identification (see sections 3.4.4 and 3.4.5.1).
– User definable panels of commands ❹ ("**PANEL** interface": see section 2.8.3) for executing frequently used command sequences (like in "STYLE GP").

Another main feature of this interface is an automatic generation of the tree command into a pulldown menu which generates a "Command Argument Panel" for each terminal command with the list and description of the parameters to be filled before command execution. The functionality of these panels is improved, with respect to the "STYLE G", especially for what concerns the dynamic parameter setting, e.g. through slide bars whenever it is desirable or possible.



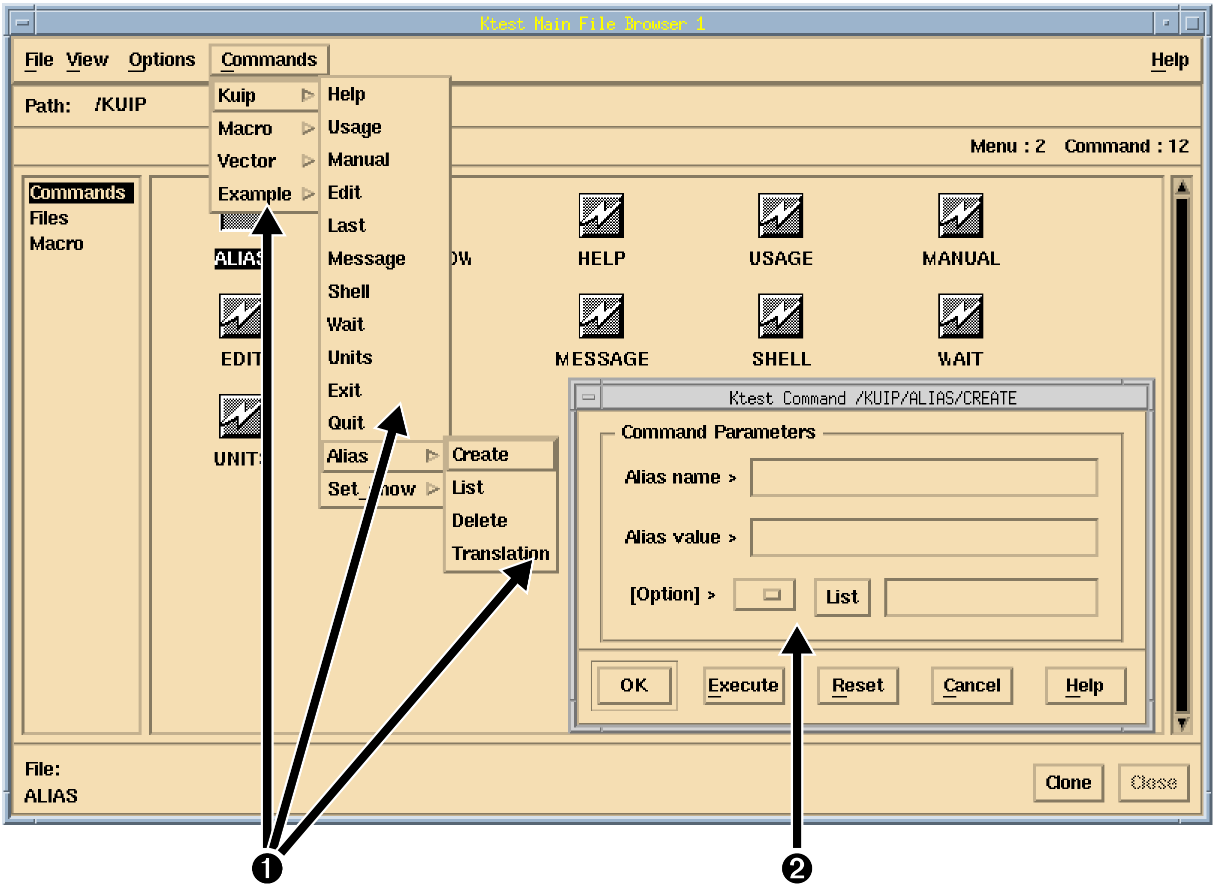
❶ **“Executive Window”**
❷ **“Main Browser”**.
❸ HIGZ **Graphics Window** (optional).
❹ User definable panel of commands (“**PANEL** interface”).

① **“Main Browser”** entry for “Commands”.
② Pull-down Menu“ Commands”.
③ **Input Pad**.

Figure 2.3: What Do You Get?

This interface offers the user several possibilities to execute a command (Fig. 2.3):

– For “experienced” users it is still possible to type the full command (with its parameters) into the **Input Pad** of the **“Executive Window”** ③. This is what the users have to do on a dumb terminal.
– The pulldown menu “Commands” ② in the top menu-bar of the **“Main Browser”** gives access to all the commands defined by KUIP or the application. When releasing the left mouse button on a terminal item of this menu the user has access to the corresponding “Command Argument Panel” (see figure 2.4). He can fill the list of parameters with some values and execute the command by pressing the “OK” or “Execute” button. (“OK” will destroy the panel afterwards).
– Another possibility is to select the “Commands” entry in the **“Main Browser”** ① and traverse the



❶ Pulldown menu "Commands" with the complete tree command structure.
❷ "Command Argument Panel" for the KUIP command /KUIP/SET_SHOW/STYLE.

Figure 2.4: Pulldown Menu Access to Commands

directory structure of commands (see section 2.8.1.1). The default action for a terminal command (double click with the left mouse button) is the command execution. It is also possible to have access to the corresponding "Command Argument Panel" by selecting the entry "Execute ..." in the popup menu displayed by a single click with the right mouse button.

– It is also possible to have access to the "Command Argument Panel" for a command by putting a "-" in front of the command (typed into the **Input Pad**).

2.2 Command line syntax

The general syntax of a *command line* is a *command path* optionally followed by an *argument list*. The command path and the arguments have to be separated from each other by one or more space characters. Therefore arguments containing spaces or other special characters have to be quoted.

In the following we want to use an appropriate formalism to describe the syntax rules. The notation will be introduced step by step as needed. The verbal explanation given above can be written as:

$$\textit{command-line} \quad ::= \quad \textit{command-path} \quad \{\ \textit{argument}\ \}$$

The *slanted* symbols are non-terminal, i.e. they are composed of other terminal or non-terminal symbols. The definition of a non-terminal symbol is denoted by "**::=**". Symbols enclosed in braces ("{...}") are optional and they can appear zero or more times.

2.2.1 Command structure

The set of commands is structured as an (inverted) tree (see figure 2.5), comparable to a Unix file system. The command set can be dynamically extended by linking new commands or menus into the tree.

Compared to a flat list structure the tree allows a cleaner representation through menus, especially when the command set is large. For example, PAW (the first application using KUIP) has more than 200 commands. It would be hard to visualize such a number of command in a single graphics menu.

2.2.1.1 Abbreviations

A command path consists of a menu path and a command name. The menu path itself consists of a list of menu names up to an arbitrarily deep level of sub-menus.

$$\textit{command-path} \quad ::= \quad [\textit{menu-path}\texttt{/}]\textit{command-name}$$
$$\textit{menu-path} \quad ::= \quad [\texttt{/}]\textit{menu-name}\{\texttt{/}\textit{menu-name}\}$$

Here we introduced two more notations. Symbols in teletype mode ("`/`") are literals, i.e. the menu and command names have to be separated by a slash character. Symbols enclosed in brackets ("[...]") are optional which can appear zero or one times.

These syntax rules already show that a command path may be abbreviated by omitting part of the leading menu path. For example, if the complete command path is

```
/MENU/SUBMENU/COMMAND
```

valid abbreviations are

```
MENU/SUBMENU/COMMAND
SUBMENU/COMMAND
COMMAND
```

but **not** "`MENU/COMMAND`" or "`/SUBMENU/COMMAND`". Note that the command name matching is case-insensitive, i.e. the following are all valid possibilities:

```
COMMAND
command
Command
```

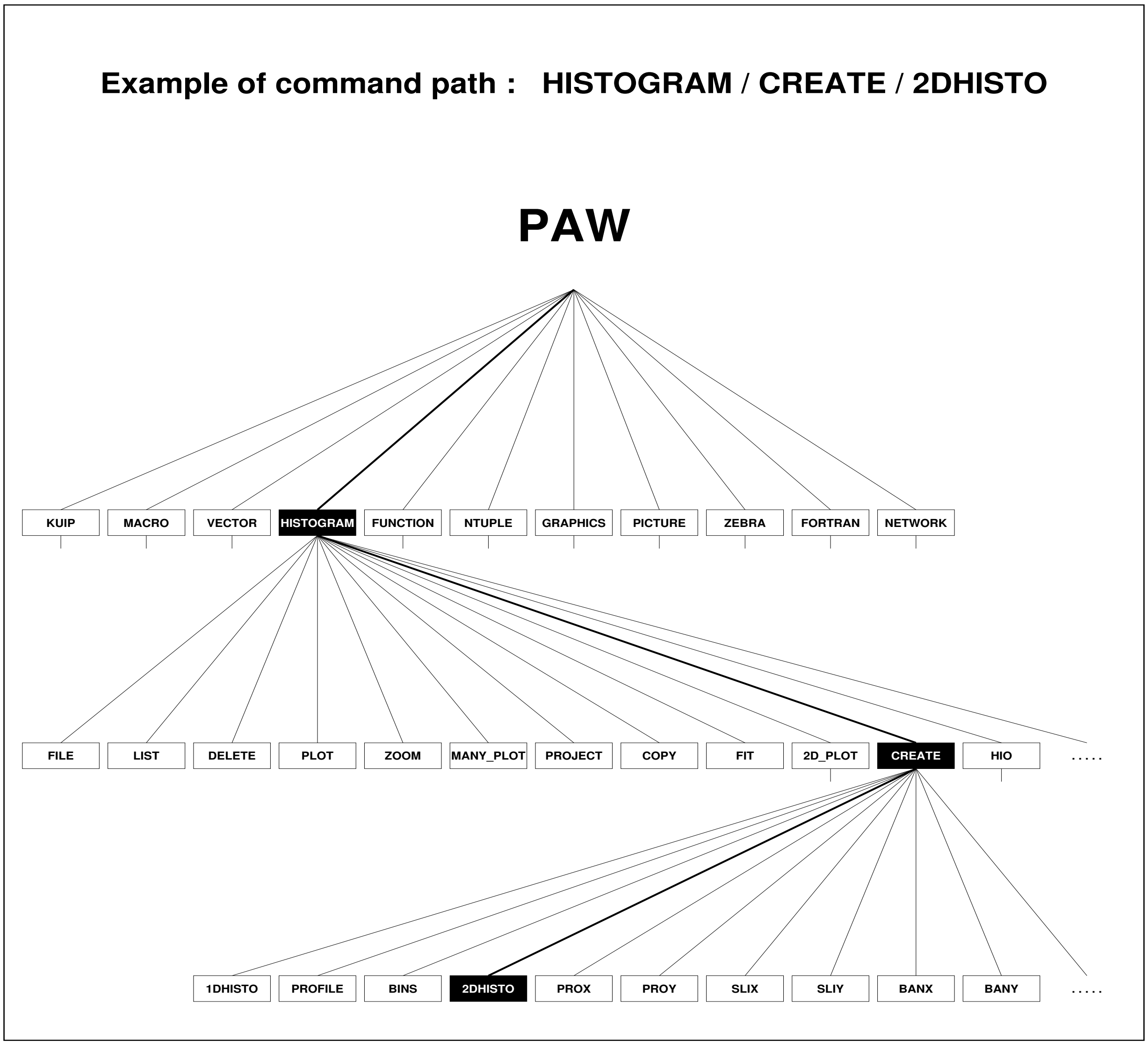


Figure 2.5: Example of the PAW command tree structure

Furthermore, menu and command names may be abbreviated by omitting trailing parts, i.e.

```
SUB/COMMAND
COMMA
/M/S/C
```

are also valid abbreviations.

The shortest unambiguous abbreviation for any command is not fixed but depends on the whole command set. KUIP lists all possible ambiguities if a given abbreviation has no unique match:

```
PAW > LIST
 *** Ambiguous command list. Possible commands are :

 /KUIP/ALIAS/LIST
 /MACRO/LIST
 /VECTOR/LIST
 /HISTOGRAM/LIST
 /NTUPLE/LIST
 /PICTURE/LIST
```

2.2.1.2 Ambiguity resolution

Abbreviations can lead to ambiguities if the abbreviation matches more than one command path. For example, in an application with the commands

```
/MENU/COMPUTE
/MENU/SUBMENU/COMMAND
/MENU/OTHERMENU/COMMA
```

typing "COM" matches all three commands and "COMM" still matches the last two.

The list of all executable commands can be obtained by just typing "/". The single slash matches every command element and therefore all available commands will be listed as possible ambiguities.

Since users tend to use abbreviations heavily also in command scripts adding a new command always risks to break these scripts by introducing a sudden ambiguity. In order to alleviate this problem a set of resolution rules apply before an abbreviation is finally considered ambiguous.

The first rule is that an exact match for the command name takes preference, i.e. "COMMA" resolves to the third command only. The second rule prefers the lowest number of menu levels. For example, "COM" resolves to the first command because the other two matches are one more menu level down.

2.2.1.3 More on command name resolution

KUIP provides additonal commands which can affect the way the command name, i.e. the first token in a command line, is interpreted.

Changing the root menu

The command SET/ROOT defines the menu from which the search for command name starts. It is not quite comparable to the Unix cd or VMS SET DEFAULT command. If no matching command is found going downwards from the SET/ROOT menu a second attempt is made starting off at the top menu "/".

The SET/ROOT command allows applications to overload names which are already used by one of the KUIP built-in commands. For example, CMZ has its own command /CMZ/EDIT clashing with the command /KUIP/EDIT. Therefore, CMZ defines

```
SET/ROOT /CMZ
```

and typing "EDIT" invokes the CMZ command while other KUIP commands can be abbreviated as usual. Note that even with SET/ROOT a typed command starting with a slash will only be searched for from the top of the command tree, i.e. "/K/ED" will not be tried as "/CMZ/K/ED".

Disabling commands

The command SET/VISIBILITY allows to disable/enable individual commands. Disabled commands cannot be executed and they do not contribute to name ambiguities. However, the HELP information is still available. In STYLE G disabled commands are shown with a grey or hatched background.

Note that the VISIBILITY command can disable itself which makes it impossible to re-enable any command.

Automatic macro execution

The command MACRO/DEFAULT implements two facilities. First it allows to define a directory search path used by the EXEC command for locating .kumac macro files. Second it controls the implicit interpretation of the command name token as a possible macro filename:

-Command	This is the default setting which does not try to interpreted cmd as macro name.
-Auto	If the search path contains a file cmd.kumac it is executed, i.e. the actual command becomes "EXEC cmd", otherwise the search for a command named cmd starts.
-AutoReverse	If cmd is either not a command name or ambiguous and a file cmd.kumac exists the command is transformed into "EXEC cmd".

Command template

The command SET/COMMAND allows to define a template which is used whenever the command token does not match any command name. The template can contain "$1", ..., "$9" which are substituted with the *n*'th token from the original command line, or "$*" which is replaced by the complete line. For example, a KUIP application can be turned into a calculator by

```
KUIP > SET/COMMAND 'mess $eval($*)'
KUIP > 17+2*5
 27
```

"SET/COMMAND 'EXEC $*'" has almost the same effect as "DEFAULT -AutoReverse" but these are two distinct facilities which can be active simultaneously. The difference is that for SET/COMMAND the token in the command name position must not match any command. If does not apply if the token is an ambiguous command name.

Both Auto/AutoReverse and SET/COMMAND logic are ignored during the execution of macro scripts.

2.2.2 Arguments

Most commands have *parameters* for which the user is expected to supply *argument values*. Parameters are either *mandatory* or *optional*. Mandatory arguments which are not specified on the command line are prompted for. If optional arguments are omitted a default value is used instead.

Mandatory parameters always precede the optional parameters. The command USAGE allows to see the number of parameters for a command:

```
KUIP > usage manual

 * KUIP/MANUAL ITEM [ OUTPUT OPTION ]
```

The optional parameters are enclosed in square brackets. The default values can be seen from the help text for a command. The STYLE command shown in figure 2.6 has only optional arguments. The corresponding default values are indicated in the help information as "D=*value*". There is also the case of optional parameters without fixed default values. For these commands the application writer has to provide an appropriate default at execution time.

```
KUIP > HELP STYLE

 * KUIP/SET_SHOW/STYLE [ OPTION SGYLEN SGSIZE SGYSPA SGBORD WKTYPE ]

   OPTION     C 'Option' D='?'
   SGYLEN     R 'max Y LENgth of each menu item box' D=0.025 R=0.005:0.25
   SGSIZE     R 'space available for the application' D=0.8 R=0:0.90
   SGYSPA     R 'max Y length of space between menus' D=0.02 R=-0.5:0.50
   SGBORD     R 'X or Y border for menus' D=0.015 R=0:0.25
   WKTYPE     I 'Graphics workstation type' D=0

   Possible OPTION values are:

    ?   show current style
    C   Command line : select Command line input
    AN  Menu with Numbers : select general Alpha menu (with Numbers)
    AL  Menu with Letters : select general Alpha menu (with Letters)
```

Figure 2.6: Parameter types, default values, and range limits

Mandatory parameters may also have a default value which is used if the prompt is acknowledged by simple hitting the RETURN-key. Otherwise the proposed default is the value used in the previous command execution.

The STYLE command also shows that there are three different kind of parameters: character values indicated by "C" after the parameter name, real values ("R") and integer values ("I").

Whether character values are case-sensitive is up to the application. The application writer has three choices to retrieve a character argument:

KUGETC returns the string converted to uppercase.

KUGETS returns the string as it was typed in.

KUGETF returns on operating systems with case-sensitive filenames (Unix) the string depending on the current setting of the FILECASE command. The string is either left as it is, or it is converted to lowercase. If filenames are not case-sensitive the argument value is converted to whatever case is required by the operating system.

Numeric (real or integer) parameters may be restricted in the range of acceptable values. In the help text this is indicated as "R=***lower***:***upper***. If the argument value is outside the range KUIP prompts the user to enter an acceptable value before the command can be executed. The lower or upper range value may be missing to indicate an unlimited range in one direction. Instead of a simple numeric value the argument may also be an expression.

For both numeric and character parameters the range may also be given as a comma-separated list of values. KUIP will accept an argument only if it matches one of the values in the list.

In general the arguments given on the command line are assigned to the command parameters from left to right but there are also ways to change the order. In our syntax notation, using "|" to indicate possible alternatives, we can write:

argument ::= *value* | `!` | `!!` | *name*`=`*value* | `-`*value*

An argument given as a simple value is assigned to the next parameter expected. The special values "`!`" and "`!!`" are templates for the default value and the value from the previous command execution, respectively.

2.2.2.1 Named arguments

The form "*name*`=`*value*" allows to invert the argument order or to skip a list of optional parameters for which the default values should be used. For example,

```
STYLE G SGBORD=0.1
```

is equivalent to

```
STYLE G ! ! ! 0.1
```

KUIP strips off the "*name*`=`" part before passing the argument values to the application. In fact the application program cannot distinguish which of these possible forms the user actually typed. A simple argument following a named argument is assigned to the parameter following the named parameter, i.e.

```
STYLE G SGBORD=0.1 1
```

is equivalent to

```
STYLE G ! ! ! SGBORD=0.1 WKTYPE=1
```

Parameter names are case-insensitive but in general they may not be abbreviated. However, the application write can allow abbreviations up to a certain minimum length. In the help text this is indicated by a "`*`" inside the parameter name. For example, if the parameter name is shown as

```
LIB*RARY
```

the acceptable abbreviations are "`LIB=`", "`LIBR=`", "`LIBRA=`", "`LIBRAR=`", and "`LIBRARY=`".

KUIP does not insist that an argument of the form "*name*`=`*value*" matches one of the parameter names. The argument including the "*name*`=`" part is simply assigned to the next parameter expected.

2.2.2.2 Option arguments

The last alternative "`-`*value*" to specify an argument applies only to *option* parameters. (Note the distinction between *option* and *optional*. Option parameters are usually but not necessarily optional.) In the help text option parameters are tagged by the list of possible values (figure 2.7). Frequently these parameters are named "`OPTION`" or "`CHOPT`".

The "`-`*value*" form allows to specify option arguments out of order, emulating the Unix style of options preceeded other command arguments. For example,

```
MANUAL -LATEX /KUIP
```

is equivalent to

```
MANUAL /KUIP OPTION=LATEX
```

```
PAW > HELP MANUAL

 * KUIP/MANUAL ITEM [ OUTPUT OPTION ]

   ITEM       C 'Command or menu path'
   OUTPUT     C 'Output file name' D=' '
   OPTION     C 'Text formatting system' D=' '

   Possible OPTION values are:

   ' '     plain text : plain text format
    LATEX  LaTeX format (encapsulated)
    TEX    LaTeX format (without header)
```

Figure 2.7: Example for option parameters

Note that this is **not** equivalent to "`MANUAL OPTION=LATEX /KUIP`". Unlike to the "`-`*value*" form subsequent simple arguments are still assigned to the next parameter expected, not to the one following the option parameter itself.

Since a leading "`-`" can be part of a valid (non-option) argument the value is checked against a set of rules before it is actually interpreted as an option assignment.

The option argument can be a concatenation of several of the allowed option values. KUIP checks that the argument string is exclusivly constructed from valid option values. This check is done by removing matches of option values from the argument string, starting with the longest option values first. For example, with the definition

```
Possible OPTION values are:

 AB
 ABC
 CD
```

the argument "`-ABCD`" is not interpreted as option assignment because after removing the longest match "`ABC`" the remainder "D" is not anymore a valid option value. (This case would have to be written as "`-CDAB`". KUIP does not check whether the combination of values is valid. It is left to the application to refuse execution, e.g. if some of the given option values are mutually exclusive.)

Even with this consistency check there is still a problem arising for commands using digits as option values. One example is the PAW command `SMOOTH` (figure 2.8). The command line

```
SMOOTH -1 2
```

could be interpreted as

```
SMOOTH ID=2 OPTION=-1
```

Since histogram identifiers can have the form of a negative number the desired interpretation is the natural order

```
SMOOTH ID=-1 OPTION=2
```

The application writer has to inform KUIP about this by giving the `ID` parameter the "`Minus`" attribute. For numeric parameters the "`Minus`" attribute is implicit. However, the argument is taken as an option assignment if the parameter has a limited range which does not include the corresponding negative value. For example,

```
SMOOTH 10 SENSIT=2 -1
```

is interpreted as

```
SMOOTH ID=-1 OPTION=1 SENSIT=2
```

since "-1" is outside the range for the SMOOTH parameter.

```
* HISTOGRAM/OPERATIONS/SMOOTH ID [ OPTION SENSIT SMOOTH ]

  ID         C 'Histogram or Ntuple Identifier' Minus
  OPTION     C 'Options' D='2M'
  SENSIT     R 'Sensitivity parameter' D=1. R=0.3:3.
  SMOOTH     R 'Smoothness parameter' D=1. R=0.3:3.

  Possible OPTION values are:

   0  Replace original histogram by smoothed.
   1  Replace original histogram by smoothed.
   2  Store values of smoothed function and its parameters without replacing
      the original histogram (but see note below) - the smoothed function can
      be displayed at editing time - see HISTOGRAM/PLOT.
   M  Invoke multiquadric smoothing.
```

Figure 2.8: HELP SMOOTH

```
* HISTOGRAM/PLOT [ ID CHOPT ]

  ID         C 'Histogram Identifier' Loop Minus
  CHOPT      C 'Options' D=' ' Minus

  Possible CHOPT values are:

  ' '      Draw the histogram.
   C       Draw a smooth curve.
   S       Superimpose plot on top of existing picture.
   +       Add contents of ID to last plotted histogram.
   -       Substract contents of ID to last plotted histogram.
```

Figure 2.9: HELP HISTO/PLOT

The "-" in an option assignment is usually stripped off before the value is passed to the application program. The exception is if the minus sign itself is one of the valid option values and the next argument expected belongs to the option parameter itself. Consider the PAWcommand HISTO/PLOT (figure 2.9). The command line

```
H/PLOT -S 1
```

is interpreted as

```
HISTO/PLOT ID=1 CHOPT=S
```

while

```
H/PLOT 1 -S
```

is equivalent to

```
HISTO/PLOT ID=1 CHOPT=-S
```

2.2.2.3 Argument values

Since in command line blanks are used to separate the command name and the individual arguments string values containing blanks have to be quoted. The rules are the same as used by Fortran: the quote character is the apostroph "'", and apostroph inside a quoted string have to be duplicated:

```
MESS 'Hello world'
MESS 'Do or don''t'
```

The enclosing quote characters are stripped off before the argument value is passed to the application, even if they are redundant, i.e. the two forms

```
MESS 'Hello'
MESS Hello
```

are equivalent. Note that the `MESSAGE` command has only a single parameter:

```
 * KUIP/MESSAGE [ STRING ]

   STRING     C 'Message string' D=' '
...
```

Nevertheless, in most cases quoting the message string is not necessary. If the command line contains more arguments than there are parameters the additional values are concatenated to the argument for the last parameter. In the concatenation each value is separated by a (single) blank character, i.e. the commands

```
MESS 'Hello World'
MESS  Hello World
MESS  Hello       World
```

yield all the same output. Therefore the message text only needs quoting if the words should be separated by more than one space character.

Quoting inhibits the interpretation of the enclosed string as special argument values. Printing an exclamation mark as message text has to written as

```
MESS '!'
```

because "`MESS !`" would mean to take the default value for the parameter `STRING` and yield an empty line only.

Another instance is if an argument of the form "*name*=*value*" should be taken literally. For example, the command line

```
EXEC mac foo=bar
```

initializes the macro variable "`foo`" to the value "`bar`". However, if the intention is to pass the string "`foo=bar`" as argument to the macro quotes must be used:

```
EXEC mac 'foo=bar'
```

In addition, some PAWcommands, e.g.

```
* NTUPLE/PLOT IDN [ UWFUNC NEVENT IFIRST NUPD OPTION IDH ]
```

use the form "*name=value*" for equality tests in the cut expression `UWFUNC`. For example, the command

```
NT/PLOT 10.energy year=1993
```

selects all event for which the Ntuple column `YEAR` has the value `1993`. Any name clash between the Ntuple column and one of the command parameters requires quoting. If the column was called `NUPD` instead of `YEAR` the command would have to be written as

```
NT/PLOT 10.energy 'nupd=1993'
```

or alternatively as "`NT/PLOT 10.energy UWFUNC=nupd=1993`".

Finally, quoted strings are also exempted from any substitutions of aliases, KUIP system functions, and macro variables. For example,

```
MESS 'foo'
```

always prints "`foo`" while

```
MESS foo
```

can result in "`bar`" if preceded by the command "`ALIAS/CREATE foo bar`". Since square brackets denote macro variable substitution and system functions names start with a dollar-sign it is especially recommended to quote VMS file specifications.

The operator "`//`" allows to concatenate several parts to a single argument value. Unquoted strings on either side of the concatenation operator are implicitly treated as literals unless they are subject to a substitution, i.e. the command lines

```
MESS 'abc'//'def'
MESS 'abc'//def
MESS abc//'def'
MESS abc//def
MESS abcdef
MESS 'a'//'b'//'c'//'d'//'e'//'f'
```

are all equivalent (provided that `abc` and `def` are not defined as aliases). The character sequence "`//`" at the beginning or end of an argument is taken literally, e.g. in

```
CD //LUN2//1
```

the command receives the value "`//LUN21`".

2.2.3 More on command lines

The command line syntax allows to write several commands in one line and also to extend commands with long argument lists over several lines.

2.2.3.1 Multiple commands on a single line

An input line presented to the KUIP command processor may contain several commands separated by "`;`". The commands are executed sequentially as if they were on separate lines:

```
MESS Hello world!; MESS How are you?
```

is equivalent to

```
MESS Hello world!
MESS How are you?
```

Note that the text following the semicolon will not be used to satisfy any prompts emitted by the preceeding command, e.g. "`usage; manual`" will not behave as "`usage manual`".

The semicolon is **not** interpreted as line separator if it is immediately followed by a digit or one of the characters

```
  +  - *  ?  [
```

For example, issuing a VMS command with a file version number such as

```
SHELL delete *.tmp;*
```

does not require quoting. Note that this exception rule applies independently of the operating system. In order to avoid surprises we recommend to put always at least one blank after a semicolon intended to be a line separator.

Each command execution returns a status code which is zero for success and non-zero for failure. The sequences "`;&`" and "`;!`" allow to execute the remaining part of an input line depending on the status code of the preceeding command. With

```
cmd1 ;& cmd2 ; cmd3
```

the commands `cmd2` and `cmd3` are only executed if `cmd1` succeeded while with

```
cmd1 ;! cmd2 ; cmd3
```

the remaining commands are only executed if the first one failed. Note that the two characters must follow each other immediately without intervening blank.

In some commands, for example `HISTO/PLOT`, one of the parameters is marked in the help text with the attribute "`Loop`". If the corresponding argument is a comma-separated list of values KUIP implicitly repeats the command for each value in the list individually:

```
HISTO/PLOT 10,20,30
```

is equivalent to

```
HISTO/PLOT 10
HISTO/PLOT 20
HISTO/PLOT 30
```

Note that "`,`" inside parentheses is not taken as value separator, i.e.

```
HISTO/PLOT 10(1:25,1:25)
```

executes a single command.

2.2.3.2 Single commands on multiple lines

For commands with very long argument lists it can become necessary to continue it on the next line. An input line ending with an "_" character is joined with the following line.

In the concatenation the underscore itself and all but one of the leading blanks from the next line are removed. Blanks preceding the underscore are left intact. For example,

```
ME_
SS _
'Hello_
        world'
```

is an extravagant way of writing

```
MESS 'Hello world'
```

Note that the interpretation of "_" as line continuation cannot be escaped. If the command line should really end with an underscore the last argument must be quoted.

2.2.3.3 Recalling previous commands

The command lines types during a session are written into a history file. By default the file is called `last.kumac` and is updated every 25 commands. The commands `LAST` and `RECORDING` allow to change the file name and the frequency. At the start of a new session the existing file is renamed into `last.kumacold` (except on VMS) before the new `last.kumac` is created. Comment lines indicate the date and time at which the sessions were started and stopped.

In this way the user can keep track of all commands entered in the previous and in the current session. The command "`LAST -99`" flushes the buffered lines into `last.kumac` and envokes the editor on the file. The user can then extract the interactively typed commands and copy them into another `.kumac` file from which they can be re-executed.

The command "`LAST -`***n***" prints the last *n* commands entered. On a workstation this allows to re-execute command sequences by doing cut-and-paste operations with the mouse.

KUIP provides a mechanism similiar to the one found in the Unix `csh` shell for re-executing commands:

`!-`***n***	executes the *n*'th last command once more.
`!!`	is an short-cut for "`!-1`" re-executing the last command.
`!`***n***	re-executes the *n*'th command entered since the beginning of the session.
`!`	prints the commands together with their numbers. The number of lines printed depend on the recording frequency.
`!`***foo***	re-executed the latest command line starting with the string "*foo*".

The command line numbering can also be seen if the prompt string contains "`[]`":

```
KUIP > PROMPT 'Kuip[] '
Kuip[2]
```

On Unix and VMS KUIP also provides recalling and editing of command lines for re-executing. The command `RECALL` allows to choose between different key-bindings:

- Recall style KSH has an Emacs-like binding (table 2.1) similar to the one used by the ksh and bash shells. If the terminal returns ANSI escape sequences the arrow keys can be used instead of ^B/^F/^N/^P.
- Recall style DCL implements the key-binding of VMS line editing (table 2.2).
- The style names KSHO and DCLO allow to switch to overstrike mode instead of the default insert mode.
- Recall style NONE directs KUIP to do plain reading from the terminal input.

Although the default setting depends on the operating system both styles can be used on Unix and VMS. Style NONE is recommendable on systems which do swapping instead of paging. For example, on a Cray-X/MP KUIP line-editing requires that the application program itself has to react to each individual keystroke.

On Apollo/DomainOS KUIP starts up in style NONE, if the program runs in a Display Manager pad, and in style KSH otherwise. However, if crp is used from within a DM pad to run the program on a remote node the automatic identification fails and style NONE must be selected manually.

^A/^E	Move cursor to beginning/end of the line.
^F/^B	Move cursor forward/backward one character.
^D	Delete the character under the cursor.
^H, DEL	Delete the character to the left of the cursor.
^K	Kill from the cursor to the end of line.
^L	Redraw current line.
^O	Toggle overwrite/insert mode. Text added in overwrite mode (including yanks) overwrites existing text, while insert mode does not overwrite.
^P/^N	Move to previous/next item on history list.
^R/^S	Perform incremental reverse/forward search for string on the history list. Typing normal characters adds to the current search string and searches for a match. Typing ^R/^S marks the start of a new search, and moves on to the next match. Typing ^H or DEL deletes the last character from the search string, and searches from the starting location of the last search. Therefore, repeated DEL's appear to unwind to the match nearest the point at which the last ^R or ^S was typed. If DEL is repeated until the search string is empty the search location begins from the start of the history list. Typing ESC or any other editing character accepts the current match and loads it into the buffer, terminating the search.
^T	Toggle the characters under and to the left of the cursor.
^U	Kill from the prompt to the end of line.
^Y	Yank previously killed text back at current location. Note that this will overwrite or insert, depending on the current mode.
TAB	By default adds spaces to buffer to get to next TAB stop (just after every 8th column).
LF, CR	Returns current buffer to the program.

Table 2.1: Key-binding for recall style KSH

`BS/^E`	Move cursor to beginning/end of the line.
`^F/^D`	Move cursor forward/backward one character.
`DEL`	Delete the character to the left of the cursor.
`^A`	Toggle overwrite/insert mode.
`^B`	Move to previous item on history list.
`^U`	Delete from the beginning of the line to the cursor.
`TAB`	Move to next `TAB` stop.
`LF, CR`	Returns current buffer to the program.

Table 2.2: Key-binding for recall style `DCL`

2.3 Aliases

KUIP aliases allow the user to define abbreviations for parts of a command line. There are two types of aliases, *command aliases* and *argument aliases*, which differ in the way they are recognized in a command line. Both alias types can be defined by the `ALIAS/CREATE` command:

```
* KUIP/ALIAS/CREATE NAME VALUE [ CHOPT ]

  NAME      C 'Alias name'
  VALUE     C 'Alias value'
  CHOPT     C 'Option' D='A'

  Possible CHOPT values are:

   A  create an Argument alias
   C  create a Command alias
   N  No alias expansion of value
```

The alias value may be any string but the alias name can only consist letters, digits, “`_`”, “`-`”, “`@`”, and “`$`” characters. Command and argument aliases share the same name space. If a command alias with the same name as an existing argument alias is created, the argument alias is deleted first, and vice versa.

2.3.1 Argument aliases

If an argument alias name is recognized anywhere in the command line it is substituted by its value. The name matching is case-insensitive and the substitution is literally, i.e. without case folding or insertion of additional blanks. The replacement is scanned for further occurrences of alias names which in turn will be replaced as well.

The alias name must be separated from the rest of the command line either by a blank or by one of the special characters

```
/ , = : ; . % ' ( )
```

(not necessarily the same character on both sides). For example, if `foo` and `bar` are alias names, `foot` and `Bar-B-Q` are not affected. If two alias replacements need to be concatenated the “`//`” operator can be used, i.e.

```
ALIAS/CREATE DIR disk$user:[paw]
ALIAS/CREATE FIL file.dat
HISTO/FILE 1 DIR//FIL
```

translates into “HISTO/FILE 1 disk$user:[paw]file.dat”. Since argument aliases are also recognized in the command position with the definition abbreviations like HISTO/FIL cannot be used anymore.

Alias substitution does not take place inside quoted strings. The ALIAS commands themselves are treated as a special case. In the command line parsing they are specifically exempted from alias translation in order to allow aliases can be deleted and redefined without quoting. For example,

```
KUIP > ALIAS/DELETE *
KUIP > ALIAS/CREATE foo bar
KUIP > ALIAS/CREATE bar BQ
KUIP > ALIAS/CREATE foo tball
KUIP > ALIAS/LIST
 Argument aliases:
 BAR        => BQ
 FOO        => tball
 No Command aliases defined.
```

redefines FOO rather than creating a new alias name BQ. The value part, however, is subject to alias translations. If the aliases are created in reverse order

```
KUIP > ALIAS/DELETE *
KUIP > ALIAS/CREATE bar BQ
KUIP > ALIAS/CREATE foo bar
KUIP > ALIAS/LIST
 Argument aliases:
 BAR        => BQ
 FOO        => BQ
 No Command aliases defined.
```

the second alias is created as “ALIAS/CREATE foo BQ”. In this case quoting the alias value does not avoid the translation. Writing instead

```
ALIAS/CREATE foo 'bar'
```

will yield the same result. Since the ALIAS commands bypass part of the command line parsing the translation of the value part has to be applied by the ALIAS/CREATE command itself. At that stage the information about quoting is no longer available.

The option “N” allows to inhibit the alias expansion in the value. Using this option can lead to an infinite recursion of alias translations which will be detected only when one the alias names involved is actually used.

```
KUIP > ALIAS/DELETE *
KUIP > ALIAS/CREATE foo bar
KUIP > ALIAS/CREATE -N bar foo
KUIP > ALIAS/LIST
 Argument aliases:
 BAR        => foo
 FOO        => bar
 No Command aliases defined.
```

```
KUIP > foo
 *** Recursive command alias in foo
 *** Recursive argument alias in foo
 *** Unknown command: foo

KUIP > bar
 *** Recursive command alias in bar
 *** Recursive argument alias in bar
 *** Unknown command: bar
```

Alias substitution happens before the command line is split-up into command name and arguments. Hence, aliases can represent several arguments at once. For example,

```
ALIAS/CREATE limits '100 -1.57 1.57'
FUN1 10 sin(x) limits
```

is equivalent to

```
FUN1 10 sin(x) 100 -1.57 1.57
```

The quotes in the `ALIAS/CREATE` command are necessary but they are not part of the alias value. If an alias value containing blanks is supposed to be treated as a single argument four extra quotes are needed in order that

```
ALIAS/CREATE htitle '''X vs. Y'''
1D 10 htitle 100 0 1
```

is equivalent to

```
1D 10 'X vs. Y' 100 0 1
```

Argument aliases can lead to unexpected interpretations of command lines. For example, a user defining

```
ALIAS/CREATE e EDIT
```

wants “E” to be short-hand for the command `EDIT`. However, the following consequence is probably not intended:

```
PAW > nt/plot 30.e
 ***** Unknown name ---> EDIT
```

For historic reasons the default option for the `ALIAS/CREATE` command is to define an argument alias. However, the use of argument aliases can lead to subtle side-effects and should therefore be restricted as much as possible.

2.3.2 Command aliases

This problem described above does not arise if a command alias is created instead:

```
ALIAS/CREATE -C e EDIT
```

Command aliases are only recognized if they appear at the beginning of a command line (ignoring leading blanks). Hence, there is no need to protect command arguments from inadvertent substitutions. Furthermore the match must be exact (ignoring case differences), i.e. the command

```
/GRAPHICS/HPLOT/ERRORS
```

can still be abbreviated as HPLOT/E.

Alias values can also represent several commands by using one of the line separators described in section 2.2.3.1, e.g.

```
ALIAS/CREATE -C ciao 'MESS Hello world! ; MESS How are you?'
```

2.4 System functions

KUIP provides a set of built-in, so-called system functions which allow, for example, to inquire the current dialogue style or to manipulate strings. An application may provide additional functions. The complete list of available functions can be obtained from "HELP FUNCTIONS".

The function name is preceded by a $-sign. Arguments are given as a comma separated list of values delimited by "(" and ")". The arguments may be expressions containing other system functions.

Functions without arguments must be followed by a character which is different from a letter, a digit, an underscore, or a colon[1]. "$OSMOSIS" will not be recognized as the function "$OS" followed by "MOSIS". If that is the desired effect the concatenation operator has to be used: "$OS//MOSIS". Note however that two functions can follow each other, e.g. "OSMACHINE" because the $-sign does not belong to the function name.

Depending on the setting of the SET/DOLLAR command the name following the $-sign may also be an environment variable[2]. The replacement value for "$*xxx*" is obtained in the following order:

1 If *xxx* is a system function followed by the correct number and types of arguments, replace it by its value.
2 Otherwise if *xxx* is an argument-less system functions, replace it by its value.
3 Otherwise if *xxx* is a defined environment variable, replace it by its value.
4 Otherwise no replacement takes place.

[1] Excluding the colon as separator avoids the substitution of VMS logical name containing a dollar-sign such as in "DISK$OS:[dir]file.dat"

[2] On VMS there is a distinction between lowercase and uppercase names. Uppercase names (without the $-sign) are searched for first in the logical name tables and then in the symbol table while lowercase names are searched for only in the symbol table. The names HOME, PATH, TERM, and USER have a predefined meaning. In order to avoid conflicts with DCL symbols which are merely defined as abbreviations for running executables and DCL procedures all values starting with a "$" or "@" character are excluded from substitution.

2.4.1 Inquiry functions

2.4.1.1 Style inquiries

– `$STYLE` returns the name of the currently active dialogue style ("C", "G", "GP", etc.). This allows, for example, to a common logon macro containing different default setups depending whether the application is started in command line mode or in Motif mode:

```
IF $STYLE='XM' THEN
   ...
ELSE
   ...
ENDIF
```

– `$LAST` returns the previously executed command sequence:

```
KUIP > MESS Hello world! ; MESS How are you?
 Hello world!
 How are you?
KUIP > MESS $LAST
 MESS Hello world! ; MESS How are you?
```

– `$KEYVAL` returns the content of the last selected panel box in style `GP` and
– `$KEYNUM` returns row/column address in the form "*row*.*col*". The column address is always given as a two-digit number. For example, in the state shown in figure2.2 the result would be

```
PAW > MESS $KEYNUM $KEYVAL
 2.04 Histo/Plot-
```

2.4.1.2 Alias inquiries

– `$ANUM` returns the number of *argument* aliases currently defined.
– `$ANAM(`***n***`)` returns the name and
– `$AVAL(`***n***`)` returns the value of the *n*'th argument alias. No substitution takes place if *n* is not a number between 1 and `$ANUM`. There is no guarantee that "`$ANAM($ANUM)`" refers to the most recently created alias.

2.4.1.3 Vector inquiries

– `$NUMVEC` returns the number of vectors currently defined.
– `$VEXIST(`***name***`)` returns a positive number if a vector *name* is currently defined. The actual value returned is undefined and may even change between tests on the same *name*. If the vector is undefined the value "0" is returned.
– `$VDIM(`***name,dim***`)` returns the vector size along index dimension *dim*; $dim = 1$ is used if the second argument is omitted. If the vector is undefined the value "0" is returned.
– `$VLEN(`***name***`)` returns for a 1-dimensional vector the index of the last non-zero element. For 2- and 3-dimensional vectors the result is the same as for `$VDIM`. If the vector is undefined the value "0" is returned.

```
PAW > V/CREATE v1(10) R 1 2 3 4 0 6
PAW > MESS $VDIM(v1) $VLEN(v1)
 10 6
PAW > V/CREATE v2($VLEN(v1))
PAW > MESS $VDIM(v2) $VLEN(v2)
 6 0
```

2.4.1.4 Environment inquiries

- `$ARGS` returns the program arguments with which the application was invoked.
- `$DATE` returns the current date in the format "*dd*/*mm*/*yy*".
- `$TIME` returns the current time in the format "*hh*/*mm*/*ss*".
- `$RTIME` returns the number of seconds elapsed since the previous usage of `$RTIME`.
- `$CPTIME` returns the seconds of CPU time spent since the previous usage of `$CPTIME`.
- `$OS` returns an identification for the operating system the application is running on, e.g. "`UNIX`", "`VM`", or "`VMS`".
- `$MACHINE` returns an identification for the particular hardware platform or Unix brand, e.g. "`HPUX`", "`IBM`", or "`VAX`". Table 2.3 shows the `$OS` and `$MACHINE` values for the different platforms. On Unix platforms the operating system version can be obtained by `$SHELL('uname -r')`.
- `$PID` returns the process number or "`1`" if the operating system does not support the notion of process IDs.
- `$IQUEST(`***i***`)` returns the *i*'th component of the status vector

  ```
  COMMON /QUEST/ IQUEST(100)
  ```

  `IQUEST(1)` always contains the return code of the most recently executed command.
- `$DEFINED(`***name***`)` returns *name* if a variable of that name is defined, or the empty string if the variable is not defined. The argument can contain "`*`" as wildcard matching any sequence of characters. All matching variable names are returned as a blank separated list.
- `$ENV(`***name***`)` returns the value of the environment variable *name*, or the empty string if the variable is not defined.
- `$FEXIST(`***filename***`)` returns "`1`" if the file exists, or "`0` otherwise.
- `$SHELL(`***command,n***`)` returns the *n*'th line of output from the shell command.
- `$SHELL(`*command,sep*`)` returns the output from the shell command, where newlines are replaced by the separator string. The *sep* argument can be omitted and defaults to a single blank character. The `$SHELL` function is operational only on Unix systems. The *command* string is passed to the shell set by the `HOST_SHELL` command. Alias definitions etc. in the shell specific startup script (e.g. `.cshrc`) are taken into account.

2.4.2 String manipulations

- `$LEN(`***string***`)` returns the number of characters in *string*.
- `$INDEX(`***string,substring***`)` returns the position of the first occurence of *substring* inside *string* or zero if there is none.
- `$LOWER(`***string***`)` and
- `$UPPER(`***string***`)` return the argument *string* converted to lower or upper case, respectively.

`$OS`	`$MACHINE`	Platform
`UNIX`	`ALPHA`	DEC Alpha OSF
`UNIX`	`APOLLO`	HP/Apollo DomainOS
`UNIX`	`CONVEX`	Convex
`UNIX`	`CRAY`	Cray Unicos
`UNIX`	`DECS`	DECstation Ultrix
`UNIX`	`HPUX`	HP/UX
`UNIX`	`IBMAIX`	AIX for IBM/370
`UNIX`	`IBMRT`	AIX for RS/6000
`UNIX`	`LINUX`	Linux for PCs
`UNIX`	`NEXT`	NeXT
`UNIX`	`SGI`	Silicon Graphics Irix
`UNIX`	`SOLARIS`	Sun Solaris
`UNIX`	`SUN`	SunOS
`VM`	`IBM`	VM/CMS for IBM/370
`MVS`	`IBMMVS`	MVS for IBM/370
`VMS`	`ALPHA`	VMS for Alpha
`VMS`	`VAX`	VMS for Vax
`MSDOS`	`IBMPC`	MSDOS for PCs
`WINNT`	`ALPHA`	Windows/NT for DEC Alpha
`WINNT`	`IBMPC`	Windows/NT for PCs

Table 2.3: Platform identification with `$OS` and `$MACHINE`

- `$SUBSTRING(`***string*,*k*,*n***`)` returns the substring
  - *string*(k:k+n-1) if k>0, or
  - *string*(l+k+1:l+k+n) if $k \leq 0$, where l=`$LEN`(*string*).

  In any case the upper bound is clamped to `$LEN`(*string*). The argument n may be omitted and the result will extend to the end of *string*. Character counting starts with 1; by definition the replacement is empty if k=0 or n=0. If n<0 an error message is emitted.

```
KUIP > MESS $SUBSTRING(abcde,2)/$SUBSTRING(abcde,2,3)
 bcde/bcd
KUIP > MESS $SUBSTRING(abcde,-2)/$SUBSTRING(abcde,-4,3)
 de/bcd
```

- `$WORDS(`***string*,*sep***`)` returns the number of words in *string* separated by the *sep* character. Leading and trailing separators are ignored and strings of consecutive separators count as one only. The second argument may be omitted and defaults to blank as the separator character.

```
KUIP > MESS $WORDS(',abc,def,,ghi',',')
 3
```

- `$WORD(`***string*,*k*,*n*,*sep***`)` returns n words starting from word k. The last two arguments may be omitted default to blank as separator character and the replacement value extending to the last word in *string*.

```
KUIP > MESS $WORD('abc def ghi',2)
 def ghi
KUIP > MESS $WORD('abc def ghi',2,1)
 def
```

– `$QUOTE(`***string***`)` returns a quoted version of *string*, i.e. the string is enclosed by quote characters and quote characters inside *string* are duplicated. The main use of this function is if an alias value containing blanks should be treated as a single lexical token in a command line:

```
ALIAS/CREATE htitle 'Histogram title'
1d 10 $QUOTE(htitle) 100 0 1
```

Another useful application of `$QUOTE` is to pass the value of an alias or macro variable as a character constant to a COMIS function, for example

```
foo = 'bar'
CALL fun.f($QUOTE([foo]))
```

is equivalent to "`CALL fun.f('bar')`". Since the quotes around "`'bar'`" are not part of the variable value the construct "`CALL fun.f([foo])`" would given the desired result only if the value contains blanks forcing the implicit quoting in the variable substitution.

– `$UNQUOTE(`***string***`)` returns a *string* with enclosing quote characters removed. The main use of this function is if a macro variable should be treated as several blank-separated lexical tokens:

```
limits = '100 0 1'
1d 10 'Histogram title' $UNQUOTE([limits])
```

2.4.3 Expression evaluations

– `$EXEC(`***cmd***`)` executes a macro command and returns the macro's `EXITM` value. Thus

```
mess $EXEC('mname 5')
```

is equivalent to

```
EXEC mname 5
mess [@]
```

– `$EVAL(`***expr***`)` returns the value of a numeric expression. The expression can contain numeric constants and references to vector elements joined by "`+`", `-`", "`*`", "`/`". Parentheses may be used to override the usual operator precedence. In addition, the functions `ABS(`x`)` (absolute value), `INT(`x`)` (truncation towards zero), and `MOD(`x`,`y`)` (modulus) are available. Note that all operations, including division of two integer numbers, use floating point arithmetic.

```
KUIP > V/CREATE vec(3) R 1.2 3.4 4.5
KUIP > MESS $EVAL((2+3)/4) $EVAL(vec(1)+vec(2)+vec(3))
 1.25 9.1
```

Even if *expr* is merely a constant, the result is always in a canonical format with a maximum of 6 non-zero digits. Non-significant zeroes and the decimal point are omitted after rounding the last digit towards $+\infty$ or $-\infty$. A mantissa/exponent notation is used if the absolute value is $\geq 10^6$ or $< 10^-4$.

```
KUIP > MESS $EVAL(1.500) $EVAL(14.99999) $EVAL(0.000015)
 1.5 15 1.5E-05
```

The explicit use of `$EVAL` is only necessary if the result should be inserted in a place where a string is expected, for example in the `MESSAGE` command. In the instances where a command expects an integer or real argument expressions are implicitly evaluated even without the `$EVAL` function.

- `$SIGMA(`***expr***`)` passes the expression to SIGMA for evaluation. SIGMA is an array manipulation package which supports a multitude of mathematical functions (`SQRT`, `EXP`, etc.) operating on scalars and KUIP vectors:

  ```
  PAW > V/CREATE v10(10) R 1 2 3 4 5 6 7 8 9 10
  PAW > MESS $SIGMA(2*pi) $SIGMA(vsum(v10))
   6.28319 55
  ```

  For a description of the complete SIGMA expression syntax refer to the PAW manual.
  SIGMA expressions do not follow the syntax rules for KUIP expressions. Therefore they cannot contain KUIP system functions with arguments. They may, however, contain argument-less system functions, alias names, and macro variables.

- `$RSIGMA` is a slight variation of `$SIGMA`. Both functions return a scalar result in the same canonical format used by `$EVAL`. The only difference is that `$SIGMA` removes the decimal point from integral values while `$RSIGMA` leaves it in. For example, `$RSIGMA` should be used to calculate argument values to be passed to a COMIS routine

  ```
        SUBROUTINE FUN(X)
        PRINT *,X
        END
  ```

  as floating point constants:

  ```
  PAW > CALL fun.f($SIGMA(sqrt(8)))
    2.828430
  PAW > CALL fun.f($SIGMA(sqrt(9)))
    .4203895E-44
  PAW > CALL fun.f($RSIGMA(sqrt(9)))
    3.000000
  ```

If the expression evaluates to a vector result `$SIGMA` (and `$RSIGMA`) return the name of a temporary vector containing the result. `$SIGMA` with a vector result can be used in all places where a vector name is expected, e.g.

```
PAW > V/PRINT $SIGMA(sqrt(array(3,1#3)))
 ?SIG1(1) = 1
 ?SIG1(2) = 1.41421
 ?SIG1(3) = 1.73205
```

The lifetime of these vectors is limited to the current command. Hence, their names should not be assigned to macro variables and not be used in alias definitions:

```
PAW > A/CREATE square_roots $SIGMA(sqrt(array(3,1#3)))
PAW > V/PRINT square_roots
 *** VECTOR/PRINT: unknown vector ?SIG1
```

`$SIGMA` provides in principle more functionality than `$EVAL`. However, it is at the discretion of the application writer whether `$SIGMA` is actually operational (see `KUSIGM`). It requires linking with PAWLIB and may increase the size of the executable module by an unacceptable amount.

– `$FORMAT(expr,format)` returns the expression value formatted according to the Fortran *format* specifier. The possible formats are "F", "E", "G", "I", and "Z" (hexadecimal).

```
KUIP > MESS 'x = '//$FORMAT(1.5,F5.2)
 x =  1.50
KUIP > MESS 'i = '//$FORMAT(15,I5)
 i =    15
KUIP > MESS 'j = '//$FORMAT(15,I5.4)
 j =  0015
```

– `$INLINE(name)` allows to insert the value of an alias or macro variable into an expression which is then treated as being part of the expression. For example,

```
convert = '$UPPER'
foo = $INLINE([convert])('bar')
```

is equivalent to "`foo = $UPPER('bar')`", i.e. "`foo = 'BAR'`". Without `$INLINE` the content of `[convert]` would be treated as a text string with the result that "`foo = '$UPPER(''bar'')'`".

2.5 Vectors

KUIP provides optionally (`VECDEF`) the facilities to store vectors of integer or real data. These vectors, or rather arrays with up to 3 index dimensions, can be manipulated by KUIP built-in commands (see "`HELP VECTOR`"). They are also accessible to application routines (`KUGETV` and `KUVECT`). Furthermore they are interfaced to the array manipulation package SIGMA and to the Fortran interpreter COMIS (see PAW Manual[4]).

Vectors are memory resident only, i.e. they are not preserved across program executions. The commands `VECTOR/READ` and `VECTOR/WRITE` allow to save and restore vector data from an external file in text format.

Vector names may be composed of letters, digits, underscores and question marks up to a maximum length of 32 characters[3]. Names starting with "?" are reserved for internal use by KUIP.

The only exception is the predefined vector simply called "?" which has a fixed size of 100 real elements. Unlike the others the "?" vector is mapped to a fixed memory location (the common block `/KCWORK/`). It does not show up in `VECTOR/LIST` output and it is not counted in the result of `$NUMVEC`.

2.5.1 Creating vectors

Vectors can be created with the `VECTOR/CREATE` command. The size of the index dimensions is given in Fortran notation, e.g.

```
VECTOR/CREATE v1(100)
VECTOR/CREATE v2(10,10)
```

The lower index bound always starts off at 1, i.e. "`V/CREATE v(-10:10)`" is not allowed. Omitting the index dimension as in

```
VECTOR/CREATE v
```

is equivalent to

```
VECTOR/CREATE v(1)
```

[3] Vector names which should be processed by SIGMA are currently limited to 7 characters.

Definition: `VECTOR/CREATE V(NCOL)`

```
+---+---+---+---+
|   |   | * |   |    *  is addressed by  V(3)
+---+---+---+---+
```

Definition: `VECTOR/CREATE V(NCOL,NROW)`

```
+---+---+---+---+                V(:,3)  is the 1-dim array representing the 3rd row
|   |   |   |   |                 V(2,:)  is the 1-dim array representing the 2nd column
+---+---+---+---+                         the shortcut notation V(2) can be used as well
|   |   |   |   |
+---+---+---+---+
|   | * |   |   |    *  is addressed by  V(2,3)
+---+---+---+---+
```

Definition: `VECTOR/CREATE V(NCOL,NROW,NPLANE)`

```
    +---+---+---+---+
  +---+---+---+---+ |
+---+---+---+---+ | +
|   |   | * |   | + |    *  is addressed by  V(3,1,1)
+---+---+---+---+ | +
|   |   |   |   | + |
+---+---+---+---+ | +
|   |   |   |   | +
+---+---+---+---+
```

Figure 2.10: Addressing scheme for KUIP vectors

KUIP does not keep track of the actual number of index dimension given in the `VECTOR/CREATE` command, i.e.

```
VECTOR/CREATE v1(10)
VECTOR/CREATE v3(10,1,1)
```

are equivalent.

2.5.2 Accessing vectors

Single vector elements can be used in KUIP expressions where they are treated as numeric constants. Vectors with a single element only we will refer to as "*scalar vectors*". They have the special property that in expressions it is sufficient to give the name without the "`(1)`" subscript.

Complete vectors and vector subranges can be used in the `$SIGMA` function and as argument to commands expecting a vector name. The subrange notation is the same as in Fortran, e.g. `v(3:5)`. The elements of arrays are stored in column-major order, i.e. the elements `v(1,2)` and `v(2,2)` are adjacent in memory (see figure 2.10).

The vector processing commands are expected to deal only with contiguous vectors. Therefore a subrange referring to a non-contiguous set of elements is copied into a temporary vector and cannot be used as output parameter.

2.6 Expressions

KUIP has a built-in parser for different kinds of expressions: arithmetic expressions, boolean expressions, string expressions, and "garbage expressions".

expr	::=	*number*	
	|	*vector-name*	for scalar vectors
	|	*vector-name* (*expr*)	
	|	*vector-name* (*expr* , *expr*)	
	|	*vector-name* (*expr* , *expr* , *expr*)	
	|	[*variable-name*]	if variable value has form of a numeric constant or is the name of a scalar vector
	|	[*variable-name*] (*expr* ...)	if variable value is a vector name
	|	*alias-name*	if alias value has form of a numeric constant
	|	$*system-function* (...)	if function returns a numeric value
	|	- *expr*	
	|	*expr* + *expr*	
	|	*expr* - *expr*	
	|	*expr* * *expr*	
	|	*expr* / *expr*	
	|	(*expr*)	
	|	`ABS` (*expr*)	
	|	`INT` (*expr*)	
	|	`MOD` (*expr* , *expr*)	

Table 2.4: Syntax for arithmetic expressions

2.6.1 Arithmetic expressions

The syntactic elements for building arithmetic expressions are shown in table 2.4. They can be used

- in the macro statements `DO`, `FOR`, and `EXITM`
- in macro variable assignments
- as system function arguments where a numeric value is expected
- as command arguments retrieved with `KUGETI` or `KUGETR`
- as argument to the `$EVAL` function

Note that all arithmetic operations are done in floating point, i.e. "`5/2`" becomes "`2.5`". If a floating point result appears in a place where an integer is expected, for example as an index, the value is truncated.

2.6.2 Boolean expressions

Boolean expressions can only be used in the macro statements `IF`, `WHILE`, and `REPEAT`. The possible syntactic elements are shown in table 2.5.

bool	::=	*expr rel-op expr*	*rel-op*	::=	`.LT.`	|	`.LE.`
	|	*string eq-op string*		|	`<`	|	`<=`
	|	*expr eq-op string*		|	`.GT.`	|	`.GE.`
	|	`.NOT.` *bool*		|	`>`	|	`>=`
	|	*bool* `.AND.` *bool*		|	*eq-op*		
	|	*bool* `.OR.` *bool*	*eq-op*	::=	`.EQ.`	|	`.NE.`
	|	(*bool*)		|	`=`	|	`<>`

Table 2.5: Syntax for boolean expressions

In addition, a single arithmetic expression is also accepted as boolean expression, interpreting any non-zero value as *true*. This allows, for example, the short-cuts

```
IF $VEXIST(v1) THEN
...
WHILE 1 DO
...
```

instead of the explicit forms

```
IF $VEXIST(v1)<>0 THEN
...
WHILE 1=1 DO
...
```

Note, however, that an arithmetic expression is **not** equivalent to a boolean value. This implies that

```
IF $VEXIST(v1) .and. $VEXIST(v2) THEN   | error
```

is not accepted and has to be written as

```
IF $VEXIST(v1)<>0 .and. $VEXIST(v2)<>0 THEN
```

2.6.3 String expressions

String expressions can be used
- in the macro statements `CASE`, `FOR`, and `EXITM`
- in macro variable assignments
- as system function arguments where a string value is expected
- as argument to the `$EVAL` function

They may be constructed from the syntactic elements shown in table 2.6.

string	::=	*quoted-string*	
	|	*unquoted-string*	
	|	*string* `//` *string*	concatenation
	|	*expr* `//` *string*	value of expression converted to string representation
	|	`[`*variable-name*`]`	
	|	*alias-name*	
	|	`$`*system-function* `(` ... `)`	

Table 2.6: Syntax for string expressions

2.6.4 Garbage expressions

Expressions which do not satisfy any of the above syntax rules we want to call "garbage" expressions. For example,

```
s = $OS$MACHINE
```

is not a proper string expression. Unless they appear in a macro statement where specifically only an arithmetic or a boolean expression is allowed, KUIP does not complain about these syntax errors. Instead the following transformations are applied:

1 alias substitution

2 macro variable replacement; values containing a blank character are implicitly quoted

3 system function calls are replaced one by one by their value provided that the argument is a syntactically correct expression

4 string concatenation

The same transformations are also applied to command arguments. Therefore the concatenation operator "`//`" can be omitted in many cases. For example,

```
MESS $OS$MACHINE
MESS $OS//$MACHINE
MESS $EVAL($OS$MACHINE)
MESS $EVAL($OS//$MACHINE)
```

give all the same result.

2.6.5 The small-print on KUIP expressions

KUIP expressions are evaluated by a `yacc`-generated parser. `Yacc` ("*Yet Another Compiler-Compiler*") is a standard Unix tool. It produces a C routine to parse an token stream which follows the syntax rules fixed by the grammar definition.

The parser needs as front-end a lexical analyzer which reads the input stream, separates it into tokens, and returns the token type and its value to the parser. There is another Unix tool `lex` which can produce an appropriate lexical analyzer from a set of rules. In the case of KUIP the lexical analyzer had to be hand-crafted because the interpretation of a symbol depends very much on the global context. For example, if the input stream consists is simply "`foo`" the lexical analyzer has to check consecutively:

- If `foo` is defined as an alias:
  - If the alias value looks like a number, classify it as a *number*.
  - Otherwise classify the alias value as a *string*.
- Otherwise classify it as the *string* "`'foo'`".

A similar reasoning has to be applied for "`[foo]`":

- If `foo` is a defined macro variable:
  - If the variable value looks like a number, classify it as a *number*.
  - If the variable value is the name of a scalar vector, classify it as a *number*.
  - Otherwise classify the variable value as a *string*.
- Otherwise classify it as the *string* "`'[foo]'`".

KUIP macro variables do not have to (and cannot) be declared. The value is always stored as a string and it depends on the context whether the value should be interpreted as a number. Also there is no way to tell in the beginning whether the right-hand side of an assignment is an arithmetic or a string expression.

The lexical analyzer starts off interpreting tokens as a numbers if it can. For example,

```
a = '1'
b = '2'
c = [a]+[b]
```

is tokenized as "*number* + *number*" and gives "`c = 3`" even though the values assigned to `a` and `b` are originally quoted. If we have a string expression

```
[foo]//[bar]
```

this could result in the possible token sequences

```
string // string
number // string
string // number
number // number
```

depending whether the values of `foo` and `bar` look like a number. Accordingly we would have to define four grammar rules to cover these different cases. The same problem occurs in system functions expecting a string argument, e.g.

```
$SUBSTRING([foo],2,3)
```

would need two rules for `foo` being a number or a genuine string.

`Yacc` allows to avoid this inflation of necessary rules by using so-called lexical tie-ins. After having seen "`//`" or "`$SUBSTRING(`" the parser can instruct the lexical analyzer that it should not attempt to classify the next token as a number. Therefore a single rule for each system function is sufficient.

However, a lexical tie-in can only be used after the parser found a unique match between the token sequence and all grammar rules In the case of string concatenation we still have to provide two separate rules for

```
string // string
number // string
```

The grammar rule (see above) actually says that the left-hand side of the "`//`" operator can be either an arithmetic or a string expression. An arithmetic expression is evaluated and then transformed into the result's string representation. For example,

```
2*3//4
```

gives "`'64'`". On the other hand,

```
4//2*3
```

gives "'42*3'". It does not become "'46'" because the right-hand side is not consider to be an arithmetic expression. It does also not become "126" because a result of a string operation is never again treated as a number even if it looks like one.

The lexical analyzer forwards numbers in arithmetic expressions as floating point values to the parser. The result is converted back to the string representation when it has to be stored in the macro variable. Since a single numeric value already counts as an arithmetic expression the original string representation can be lost. For example,

```
a = '0123456789'
b = [a]
MESS $LEN([a]) $LEN([b])
```

results in "10 11" because the assignment "b = 0123456789" is taken as an arithmetic expression which is reformatted into 1.23457E+08. The reformatting can be inhibited by using

```
b = $UNQUOTE([a])
```

The $UNQUOTE function removes quotes around a string. If the string is already unquoted it does nothing except that in this case the parser will treat the value of [a] as a string.

Macros should not depend on this reformatting behavior. We consider it as an obscure side-effect of the present implementation rather than a feature. If it causes inconvenience and we have a good idea how to avoid it the behavior may change in a future KUIP version.

2.7 Macros

A macro is a set of command lines stored in a file, which can be created and modified with any text editor. The command EXEC invokes the macro and allows for two ways of specifying the macro name:

```
EXEC file
EXEC file#macro
```

The first form executes the first macro contained in *file* while the second form selects the macro named *macro*. The default extension for *file* is ".kumac".

Example of macro calls

```
KUIP > EXEC abc   | Execute first (or unnamed) macro of file abc.kumac
KUIP > EXEC abc#m | Execute macro M of file abc.kumac
```

In addition to all available KUIP commands the special "macro statements" in table 2.7 are valid only inside macros (except for EXEC and APPLICATION, which are valid both inside and outside).

Note that the statement keywords are fixed. Aliasing such as "ALIAS/CREATE jump GOTO" is not allowed.

2.7.1 Macro definitions and variables

A .kumac file can contain several macros. An individual macro has the form

```
MACRO macro-name  [ parameter-list ]
   statements
RETURN [ expression ]
```

Macro Statements	
STATEMENT	DESCRIPTION
`MACRO mname [ var1=val1 ... ]`	define macro `mname`
`RETURN [ value ]`	end of macro definition
`ENDKUMAC`	end of macro file
`EXEC mname [ val1 ... ]`	execute macro `mname`
`EXITM [ value ]`	return to calling macro
`STOPM`	return to command line prompt
`APPLICATION command marker`	In-line text passed to application command
`name = expression`	assign variable value
`READ var [ prompt ]`	prompt for variable value
`SHIFT`	shift numbered macro variables
`GOTO label`	continue execution at `label`
`label:`	`GOTO` target label (must terminate with a colon)
`IF expr GOTO label`	continue at `label` if `expr` is true
`IF-THEN, ELSEIF, ELSE, ENDIF`	conditional block statement
`CASE, ENDCASE`	Macro flow control
`WHILE-DO, ENDWHILE`	Macro flow control
`REPEAT, UNTIL`	Macro flow control
`DO, ENDDO`	Macro flow control
`FOR, ENDFOR`	Macro flow control
`BREAKL`	Macro flow control
`NEXTL`	Macro flow control
`ON ERROR CONTINUE`	ignore error conditions
`ON ERROR GOTO label`	continue at `label` on error condition
`ON ERROR EXITM value`	return to calling macro on error condition
`ON ERROR STOPM`	return to command input on error condition
`OFF ERROR`	deactivate the `ON ERROR GOTO` handling
`ON ERROR`	reactivate the previous `ON ERROR GOTO` setting

Table 2.7: KUIP macro statements

Each statement is either a command line or one of the macro constructs described below. For the first macro in the file the `MACRO` header can be omitted. For the last macro in the file the `RETURN` trailer may be omitted. Therefore a `.kumac` file containing only commands (like the `LAST.KUMAC`) already constitutes a valid macro.

Input lines starting with an asterisk ("`*`") are comments. The vertical bar ("`|`") acts as in-line comment character unless it appears inside a quoted string. An underscore ("`_`") at the end of a line concatenates it to the next line.

Invoking a macro triggers the compilation of the whole `.kumac` file—not just the single macro called for. The

```
ENDKUMAC
```

statement fakes an end-of-file condition during the compilation. This allows to keep unfinished material, which would cause compilation errors, simply by moving it after the ENDKUMAC statement rather than having to comment the offending lines.

The APPLICATION statement has the same form and similar functionality as the SET/APPLICATION command:

```
APPLICATION  command  marker
   text
marker
```

The text up to the next line containing only the end marker starting in the first column is written to a temporary file and then passed to the application command. The text is not interpreted in any way, i.e. variable substitution etc. does not take place.

Instead of the full spelling APPLICATION any valid abbreviation of /KUIP/SET_SHOW/APPLICATION be used, e.g. "APPL". A call to SET/APPLICATION as a result of an alias expansion, however, is not allowed.

2.7.1.1 Macro execution

Inside a macro the EXEC statement can call other macros. A macro may also call itself recursively. The EXEC command allows two different forms for specifying the macro to be executed:

```
EXEC  fname#mname  [ argument-list ]
```

and

```
EXEC  name  [ argument-list ]
```

Between the EXEC *statement* and the EXEC *command* there is a slight difference. The command "EXEC name" executes the first macro in name.kumac while the EXEC statement will try first whether a macro name is defined within the current .kumac file.

Macro execution terminates when one of the statements

```
   EXITM  [ expression ]
```

or

```
   RETURN  [ expression ]
```

or

```
   STOPM
```

is encountered. The EXITM and RETURN statements return to the calling macro. They allow to pass a return value which is stored into the special variable [@] of the calling macro. If no value is given it defaults to "0". Note that the RETURN statement also flags the end of the macro definition, i.e. the construct

```
IF ... THEN
  RETURN         | error!
ENDIF
```

is illegal. The STOPM statement unwinds nested macro calls and returns to the command line prompt immediately.

2.7.1.2 Macro variables

Macro variables do not have to be declared. They become defined by an assignment statement:

```
name = expression
```

The right-hand side of the assignment can be an arithmetic expression, a string expression, or a garbage expression (see section 2.6). The expression is evaluated and the result is stored as a string (even for arithmetic expressions).

The variable value can be used in other expressions or in command lines by enclosing the name in square brackets:

```
[name]
```

For example,

```
greet = Hello
msg = [greet]//' World'
MESS [msg]
```

If the name enclosed in brackets is not a macro variable then no substitution takes place.

Variable values can also be queried from the user during macro execution. The statement

```
READ  name  [ prompt ]
```

prompts for the variable value. If the prompt string is omitted it is constructed from the macro and variable names. The variable value prior to the execution of the READ statement is proposed as default value and will be left unchanged if the user answers simply be hitting the RETURN-key.

Macro using the READ statement

```
MACRO m
READ foo
bar = abc
READ bar
MESS [foo] [bar]
msg = ''
READ msg 'Enter message:'
MESS You said [msg].
```

Output when executing

```
KUIP > EXEC m
 Macro m: foo ? (<CR>=[foo]) 123
 Macro m: bar ? (<CR>=abc)
 123 abc
 Enter message: (<CR>=) Hello
 You said Hello.
```

2.7.1.3 Macro arguments

The EXEC command can pass arguments to a macro. The arguments are assigned to the numbered variables [1], [2], etc. For example, with the macro definition

```
MACRO m
MESS p1=[1] p2=[2]
```

we get the result

```
KUIP > EXEC m foo bar
 p1=foo p2=bar
```

Unlike named variables undefined numbered variables are always replaced by the blank string ' ', i.e.

```
KUIP > EXEC m foo
 p1=foo p2=' '
```

The MACRO statement can define default values for missing arguments. With the macro definition

```
MACRO m 1=abc 2=def
MESS p1=[1] p2=[2]
```

we get the result

```
KUIP > EXEC m foo
 p1=foo p2=def
```

The macro parameters can also be named, for example:

```
MACRO m arg1=abc arg2=def
MESS p1=[arg1] p2=[arg2]
```

Even if the parameters are named the corresponding numbered variables are created nevertheless. The named variables are a copy of their numbered counterparts rather that aliases, i.e. the above macro definition is equivalent to

```
MACRO m 1=abc 2=def
arg1 = [1]
arg2 = [2]
```

The named parameters can be redefined by a variable assignment which leaves the value of the numbered variable untouched. For example,

```
MACRO m arg=old
MESS [1] [arg]
arg = new
MESS [1] [arg]
```

yields

```
KUIP > EXEC m
 old old
 old new
```

The EXEC command allows to give values for named parameters in non-positional order. For example,

```
MACRO m arg1=abc arg2=def
MESS [arg1] [arg2]
```

can be used as

```
KUIP > EXEC m arg2=foo
 abc foo
```

Unnamed EXEC arguments following a named argument are assigned to numbered variables beyond the parameters listed in the MACRO definition. For example,

```
KUIP > EXEC m arg1=foo bar
 foo def
```

i.e. the second argument "bar" is not assigned to [arg2] or [2] but to [3]. Note that this differs from the behavior for command arguments (see section 2.2.2.1).

The construct ***name=value*** may also be used in the EXEC command for names not defined in the macro's parameter list. The variable *name* is implicitly defined inside the macro. For example,

```
MACRO m
MESS [foo]
```

yields

```
KUIP > EXEC m
 [foo]
KUIP > EXEC m foo=bar
 bar
```

Therefore a string containing a "=" must be quoted[4] if it should be passed to the macro literally:

```
KUIP > EXEC m 'foo=bar'
 foo=bar
```

Since a undefined variable name can be thought of as having the value '[name]', the construct

```
IF [var]<>'[var]' THEN
```

allows to test whether such an external variable definition was provided.

Passing a value as argument to a macro is not quite the same as assigning the value to a variable inside the macro. The macro argument is not tried to be evaluated as an arithmetic expression. String operations, however, such as concatenation and alias substitutions, are applied. For example, "EXEC m1 2*3 4//5" with

```
MACRO m1 a=0 b=0
mess [a] [b]
```

yields "2*3 45", while "EXEC m2" with

[4] Up to the 94a release of KUIP the treatment of quoted strings as macro arguments was very primitive. The value assigned to the macro variable was obtained by simply stripping off the quote character on both sides. For example, a cut expression "nation='F*'" had to be written as EXEC m 'nation='F*''

In the 94b release the quoting of macro arguments was made consistent with the general rules and the correct quoting is now EXEC m 'nation=''F*''' The old spelling is still accepted but emits a warning message Old style use of quotes in macro argument fixed to 'nation=''F*'''

```
MACRO m1
a = 2*3
a = 4//5
mess [a] [b]
```

yields “6 45”. Macro arguments are not tried as arithmetic expressions in order to allow passing of vector names without the use of quotes. Otherwise “EXEC m v1”, where v1 is a scalar vector, would pass the value of v1(1) rather than the string 'v1'.

Note that the result “6 45” can also be obtained from the first of the above examples by means of the $INLINE function:

```
MACRO m1 1=0 b=0
a = $INLINE([1])
mess [a] [b]
```

2.7.1.4 Special variables

A numbered variable cannot be redefined, i.e. an assignment such as “1 = foo” is illegal. The only possibly manipulation of numbered variables is provided by the

```
SHIFT
```

statement which copies [2] into [1], [3] into [2], etc. and discards the value of the last defined numbered variable. For example, the construct

```
WHILE [1] <> ' ' DO
  arg = [1]
  ...

  SHIFT
ENDDO
```

allows to traverse the list of macro arguments.

For each macro the following special variables are always defined:

– [0] contains the fully qualified macro file name, e.g. “./fname.kumac#mname”
– [#] contains the number of macro arguments
– [*] is the concatenation of all macro arguments separated by blanks
– [@] contains the return value of the most recent EXEC call

Like for numbered variables these names cannot be used on the left-hand side of an assignment. The values or [#] and [*] are updated by the SHIFT statement.

Example of Input Macros

```
MACRO EXITMAC
  MESSAGE At first, '[@]' = [@]
  EXEC EXIT2
  IF [@] = 0 THEN
     MESSAGE Macro EXIT2 successful
  ELSE
     MESSAGE Error in EXIT2 - code [@]
  ENDIF
RETURN
MACRO EXIT2
  READ NUM
  IF [NUM] > 20 THEN
     MESSAGE Number too large
     EXITM [NUM]-20
  ELSE
     V/CREATE V([NUM])
  ENDIF
RETURN
```

Output when executing

```
KUIP > EXEC EXITMAC
At first, [@] = 0
Macro EXIT2: NUM ? 25
Number too large
Error in macro EXIT2 - code 5
KUIP > EXEC EXITMAC
At first, [@] = 0
Macro EXIT2: NUM ? 16
Macro EXIT2 successful
```

2.7.1.5 Variable indirection and arrays

Macro variables can be referenced indirectly by constructing the name using other variables, for example

```
DO i = 1, 10
  a_[i] = [i] * [i]
ENDDO
s = 0
DO i = 1, 10
  s = [s] + [a_[i]]
ENDDO
```

While for KUIP we simply created ten variables `a_1`, ..., `a_10`, we can also look at it as an array `a_`*i*. We don't even need to remember the dimension of the array. The system function `$DEFINED` returns all defined variables matching a wildcard, for example

```
s = 0
DO i = 1, $WORDS($DEFINED('a_*'))
  s = [s] + [a_[i]]
ENDDO
```

Instead of `a_`*i* we can also use the more conventional array notation `a(`*i*`)`

```
DO i = 1, 10
  a([i]) = [i] * [i]
ENDDO
s = 0
DO i = 1, $WORDS($DEFINED('a(*)'))
  s = [s] + [a([i])]
ENDDO
```

as long as we have the possibility to match all array elements with a single wildcard expression.

Since for KUIP all array elements are just simple variables the indices do not even need to be numeric. We can also construct associative arrays where the indices are names, for example

```
events(mu) = 1000
events(el) = 100
events(tau) = 10
total = 0
names = $DEFINED('events(*)')
DO i = 1, $WORDS([names])
  name = $WORD([names],[i],1)
  total = [total] + [[name]]
ENDDO
```

By the same token we can also create multi-dimensional arrays, for example

```
DO i = 1, 3
  DO j = 1, 3
     a([i],[j]) = [i]*2+[j]
  ENDDO
ENDDO
```

The `$DEFINED` function returns the matching variable names sorted in alphabetical order, i.e.

`$DEFINED('events(*)')` is `'events(el) events(mu) events(tau)'`
`$DEFINED('a(*)')` is `'a(1) a(10) a(2) ... a(9)'`

and not necessarily in the order in which they were created.

The indirection only allows for variable substitution when constructing the actual variable name. Expression evaluation etc. does not take place and constructs such as

```
  total = [total] + [$WORD([names],[i],1)]   | invalid!
```

are not allowed.

The construct `[[name]]` can also be written as

```
[%name]
```

For example, this is another way to traverse the list of macro arguments:

```
DO i=1,[#]
  arg = [%i]
  ...
ENDDO
```

Except for the `[%name]` construct variable indirection is available only since the 95a release.

2.7.1.6 Global variables

Global variables can be made visible inside a macro by executing the commands GLOBAL/CREATE or GLOBAL/IMPORT. Technically these commands create a local variable with the same name initialized to the value of the global variable. When assigning a value to the local variable the change is also propagated to the global variable. Therefore, once they are made visible inside a macro, global variables are assigned to and used in the same way as local variables.

The GLOBAL/CREATE command creates a global variable allowing to specify an initial value and a comment text, e.g.

```
GLOBAL/CREATE m_e  0.0005 'Electron mass (GeV)'
GLOBAL/CREATE m_mu 0.106  'Muon mass (GeV)'
```

If executed inside a macro the global variable becomes visible there.

The GLOBAL/IMPORT command has an effect only when executed inside a macro. It allows to make global variables visible which have been created elsewhere. The import list may contain "*" as a wildcard for any character sequence, for example

```
GLOBAL/IMPORT m_*
```

Only those global variables existing at the time the GLOBAL/IMPORT is executed become visible. Therefore, global variables created in an inferior macro do not become visible even if they match the wildcard. For example, in

```
MACRO a
GLOBAL/IMPORT m_*
EXEC b
...
RETURN

MACRO b
GLOBAL/CREATE m_tau 1.784 'Tau mass (GeV)'
RETURN
```

m_tau is not visible in macro a unless it is imported after executing b.

Deleting a global variable in an inferior macro, on the other hand, also deletes the associated local variables in the macro call stack. For example, in

```
MACRO a
GLOBAL/IMPORT m_*
EXEC b
...
RETURN

MACRO b
GLOBAL/DELETE m_mu
RETURN
```

when returning from macro b the imported variable m_mu will become undefined.

Global variables can also be set and used from the command line, for example,

```
KUIP > g/cre x 2
KUIP > x=[x]*2
```

```
KUIP > mess [x]
 4
```

However, the implicit creation when assigning a value to an undefined variables does not apply:

```
KUIP > y=0
 *** Unknown command: y=0
```

Global variables are available only since the 95a release.

2.7.2 Flow control constructs

There are a variety of constructs available for controlling the flow of macro execution. Most for the constructs extend over several lines up to an end clause. The complete block counts as a single statement and inside each block may be nested other block statements.

The simplest form of flow control is provided by the

```
GOTO label
```

statement which continues execution at the statement following the target label:

```
label:
```

If the jump leads into the scope of a block statement, for example a `DO`-loop, the result is undefined. The target may be given as an expression evaluating to the actual label name, e.g.

```
name = label
  ...
GOTO [name]
  ...
label:
```

In the label definition the colon must follow the label name immediately without any intervening blanks. The label may be followed by a command on the same line, e.g.

```
label: MESS Hello
```

2.7.2.1 Conditional execution

```
IF expression THEN
   statements
ELSEIF expression THEN
   statements
...
ELSEIF expression THEN
   statements
ELSE
   statements
ENDIF
```

The general `IF` construct executes the statements following the first `IF`/`ELSEIF` clause for with the boolean expression is true and then continues at the statement following the `ENDIF`.

The `ELSEIF` clause can be repeated any number of times or can be omitted altogether. If none of the expressions is true, the statements following the optional `ELSE` clause are executed.

```
IF expression GOTO label
```

This old-fashioned construct is equivalent to

```
IF expression THEN
   GOTO label
ENDIF
```

```
CASE expression IN
(label)  [ statements ]
...
(label)  [ statements ]
ENDCASE
```

The `CASE` switch evaluates the string expression and compares it one by one against the label lists until the first match is found. If a match is found the statements up to the next label are executed before skipping to the statement following the `ENDCASE`. None of the statements are executed if there is no match with any label.

Each label is a string constant and the comparison with the selection expression is case-sensitive. If a label is followed by another label without intervening statements then a match of the first label will skip to the `ENDCASE` immediately. In order to execute the same statement sequence for distinct labels a comma-separated list of values can be used. The “`*`” character in a label item acts as wild-card matching any string of zero or more characters, i.e. “`(*)`” constitutes the default label.

Example for CASE labels with wild-cards

```
MACRO CASE
   READ FILENAME
   CASE [FILENAME] IN
   (*.ftn, *.for)  TYPE = FORTRAN
   (*.c)           TYPE = C
   (*.p)           TYPE = PASCAL
   (*)             TYPE = UNKNOWN
   ENDCASE
   MESSAGE [FILENAME] is a [TYPE] file.
RETURN
```

2.7.2.2 Loop constructs

The loop constructs allow the repeated execution of command sequences. For `DO`-loops and `FOR`-loops the number of iterations is fixed before entering the loop body. For `WHILE` and `REPEAT` the loop count depends on the boolean expression evaluated for each iteration.

```
DO loop = start_expr, finish_expr  [, step_expr ]
   statements
ENDDO
```

The step size defaults to "`1`". The arithmetic expressions involved can be floating point values but care must be taken of rounding errors. A `DO`-loop is equivalent to the construct

```
count = ( finish_expr - start_expr ) / step_expr
loop = start_expr
step = step_expr
label:
IF [count] >= 0 THEN
   statements
   loop = [loop] + [step]
   count = [count] - 1
   GOTO label
ENDIF
```

where all variables except for `loop` are temporary.

Note that "`DO i=1,0`" results in zero iterations and that the expressions are evaluated only once. i.e. the loop

```
n = 10
DO i=1,[n]
   MESS [i] [n]
   n = [n] - 1
ENDDO
```

is iterated 10 times and leaves "`i = 11`" afterwards.

```
FOR name IN expr_1 [ expr_2 ... expr_n ]
   statements
ENDFOR
```

In a FOR-loop the number of iterations is determined by the number of items in the blank-separated expression list. The expression list must not be empty. One by one each expression evaluated and assigned to the variable name before the statements are executed. The equivalent construct is the loop-unrolling

```
name = expr_1
statements
name = expr_2
statements
...
name = expr_n
statements
```

The expressions can be of any type: arithmetic, string, or garbage expressions, and they do not need to be all of the same type. In general each expression is a single list item even if the result contains blanks. For example,

```
foobar = 'foo bar'
FOR item IN [foobar]
   MESS [item]
ENDFOR
```

results in a single iteration. The variable [*] is treated as a special case being equivalent to the expression list "[1] [2] ... [n]" which allows yet another construct to traverse the macro arguments:

```
FOR arg IN [*]
   ...
ENDFOR
```

```
WHILE expression DO
   statements
ENDWHILE
```

The WHILE-loop is iterated while the boolean expression evaluates to true. The loop body is not executed at all if the boolean expression is false already in the beginning. The equivalent construct is:

```
label:
IF expression THEN
   statements
   GOTO label
ENDIF
```

```
REPEAT
   statements
UNTIL expression
```

The body of a REPEAT-loop is executed at least once and iterated until the boolean expression evaluates to true. The equivalent construct is:

```
label:
   statements
IF .NOT. expression GOTO label
```

```
BREAKL  [ levels ]
```

allows to terminate a loop prematurely. The BREAKL statement continues executing after the end clause of the enclosing DO, FOR, WHILE, or REPEAT block.

```
NEXTL  [ levels ]
```

allows to terminate one loop iteration and to continue with the next one. The NEXTL statement continues executing just before the end clause of the enclosing DO, FOR, WHILE, or REPEAT block.

Both BREAKL and NEXTL allow to specify the number of nesting levels to skip as an integer constant.

Example of using BREAKL and NEXTL

```
WHILE 1=1 DO
   ...
   IF expr THEN
      BREAKL
   ENDIF
   ...
   DO i=1,[#]
      ...
      IF [%i]='-' THEN
         NEXTL
      ENDIF
      IF [%i]='--' THEN
         NEXTL 2
      ENDIF
      ...
   ENDDO
   ...
ENDWHILE
```

Equivalent code using GOTOs

```
WHILE 1=1 DO
   ...
   IF expr GOTO break_while
   ...
   DO i=1,[#]
      ...
      IF [%i]='-' GOTO next_do
      IF [%i]='--' GOTO next_while
     ...
   next_do:
   ENDDO
   ...
next_while:
ENDWHILE
break_while:
```

2.7.2.3 Error handling

Each command returns a status code which should be zero if the operation was successful or non-zero if any kind of error condition occurred. The status code is stored in the `IQUEST(1)` status vector and can be tested as, for example

```
HISTO/FILE 1 foo.hbook
IF $IQUEST(1)<>0 THEN
*-- cannot open file
   ... do some cleanup
   EXITM 1
ENDIF
```

```
ON ERROR GOTO label
```

installs an error handler which tests the status code after each command and branches to the given label when a non-zero value is found. The error handler is local to each macro.

```
ON ERROR EXITM  [ expression ]
```

and

```
ON ERROR STOPM
```

are short-hand notations for an `ON ERROR GOTO` statement with a `EXITM` or `STOPM` statement, respectively, at the target label.

```
ON ERROR CONTINUE
```

nullifies the error handling. Execution continues with the next command independent of the status code. This is the initial setting when entering a macro.

```
OFF ERROR
```

and

```
ON ERROR
```

allow to temporarily suspend and afterwards reinstate the previously installed error handling. Note that the `OFF/ON` settings do not nest, for example

```
ON  ERROR EXITM
OFF ERROR        | behave like ON ERROR CONTINUE
ON  ERROR STOPM
OFF ERROR
```

```
ON   ERROR        | restore ON ERROR STOPM
ON   ERROR        | unchanged, i.e. not ON ERROR EXITM !
```

Another way of testing the status code of a command is to use the line separators "`;&`" and "`;!`" (see section 2.2.3.1). These operators take precedence over the `ON ERROR` setting.

```
cmd1 ;& cmd2 ; cmd3
```

is roughly equivalent to

```
OFF ERROR
cmd1
IF $IQUEST(1)=0 THEN
   cmd2
   ON ERROR
   cmd3
ENDIF
ON ERROR
```

except that the `ON/OFF ERROR` statements are virtual and do not overwrite the setting saved by a real `OFF ERROR` statement.

2.8 Motif mode

2.8.1 The KUIP/Motif Browser Interface

The KUIP/Motif Browser interface is a general tool to display and manipulate a tree structure of objects which are defined either by KUIP itself (commands, files, macros, etc.) or by the application (e.g. in PAW++: Zebra and Hbook files, Chains, etc.). The objects contained in the currently selected directory can be displayed in various forms: big icons, small icons, text only, etc. It is possible to perform actions on these objects or the directories it-selves by accessing pop-up menus directly attached to them: this behavior of the browser gives access to a "direct object manipulation" user interface by opposition to the usual "command mode interface". Adding your application specific objects into the browser is mainly done through the KUIP "Command Definition File" (CDF): you will not get involved in any kind of Motif programming.

2.8.1.1 Description of the "Main Browser" Window

For any application based on KUIP/Motif one browser will be automatically created and displayed: it is called the **"Main Browser"**. Later on it is possible to "clone" this browser (by pressing the corresponding button at the bottom/right) when it is in a certain state. This will give to the user the possibility to have several instances of the browser window, and look at the same time to different kind of objects.

A "browser window" is composed of (Fig. 2.11):

- A menu bar with the menu entries "File" ①, "View" ②, "Options" ③, "Commands" ④ and "Help" ⑤.
- A two lines text/label area (❶ and ❷).
- The middle part of the browser is divided into two scroll-able windows: the "FileList" or **"Browsable window"** ❸ at the left and the "DirList" or **"Object window"** ❹ at the right.
- Two lines of information at the bottom (❺ et ❻), plus a "Clone" ❽ and a "Close" ❾ buttons.

Below follows a description of the middle (and main) part of the browser which is divided into two scroll-able windows on the left and right sides (Fig. 2.11):

- The left hand "FileList" or **"Browsable window"** ❸ shows the list of all the currently connected browsables. As you will see with more detail (sections 3.3 and 3.4), a "browsable" is simply a container of objects and is defined with the "`>Browse`" directive in the CDF. The browsables "Commands", "Files" and "Macros" are built-in inside KUIP itself and are always displayed. Each application can add to this list its own definitions for any kind of browsables (e.g. in PAW++: "Zebra", "Hbook", "Chains" and "PAWC") Some browsables can also be attached at run time by selecting the corresponding "Open" entry in the menu "File" (e.g. in PAW++: ZEBRA/RZ files for access to histograms and Ntuples).
  Pressing the right mouse button in this window shows a pop-up menu with all the possible actions which have been defined for this browsable.
  Selecting one item (or browsable) in this window with the left mouse button executes by default the "List" action (first entry of the pop-up menu): it displays the content of the browsable in the right hand window ("DirList" or **"Object window"**)
  Note that the first entry of the pop-up menu of actions for one browsable is always "List" and that the last entry is always "Help" : it should give information concerning the selected browsable (see section 3.4.5.4 to know how to fill the information).

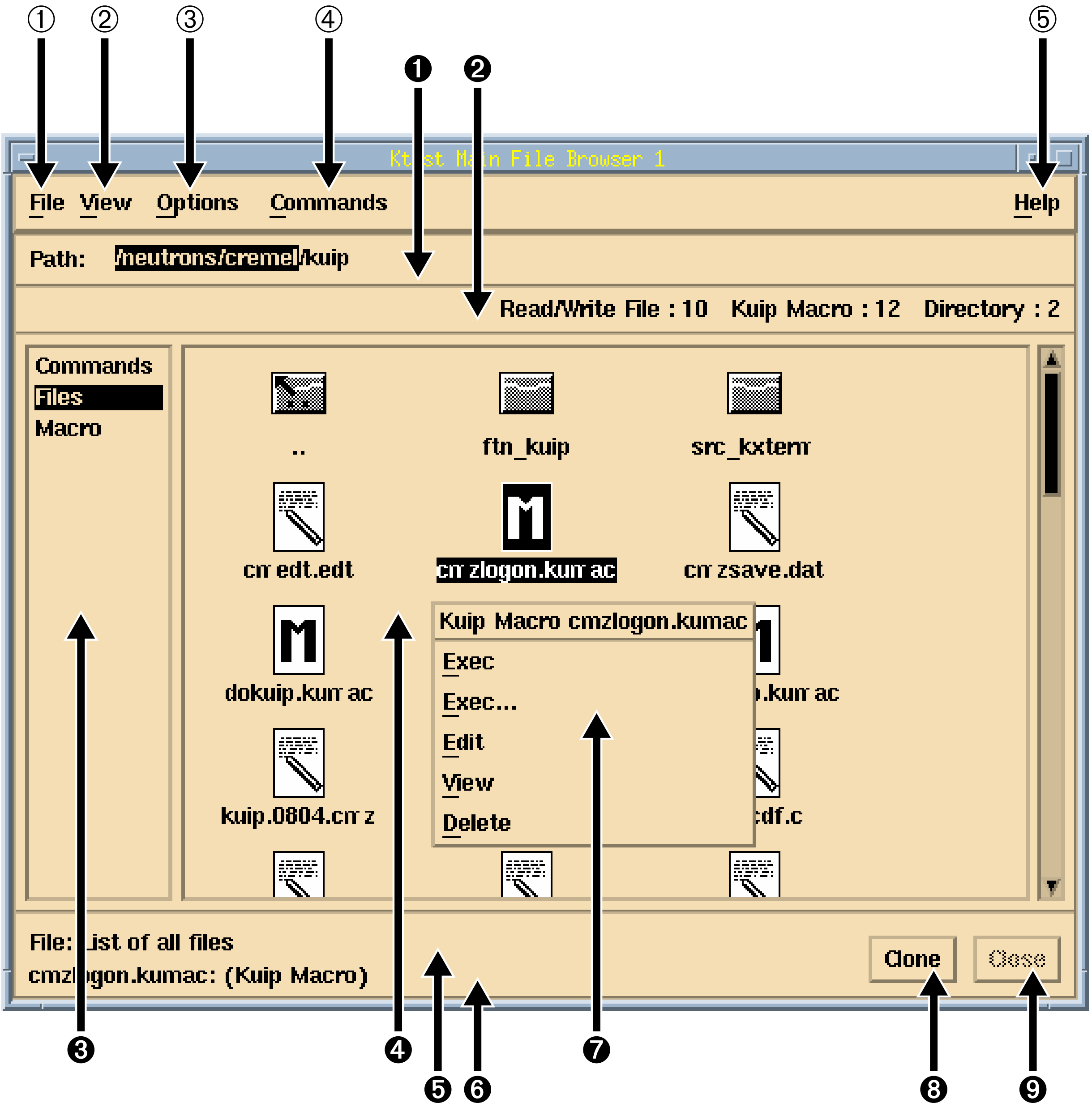


Figure 2.11: KUIP/Motif **"Main Browser"** Window

– The right hand "DirList" or **"Object window"** ❹ shows the content of the currently selected browsable for the selected path. E.g. when you select the browsable "Macro" (built-in inside KUIP), you will get all the KUIP macro files and sub-directories which are contained in the selected directory.
  Objects are selected by clicking on them with the left mouse button. Pressing the right mouse button pops up a menu of possible operations depending on the object type ❼.
  An item in a pop-up menu is selected by pointing at the corresponding line and releasing the right mouse button. Double clicking with the left mouse button is equivalent to selecting the first menu item.

Each menu item executes a command sequence where the name of the selected object is filled into the appropriate place. By default the command is executed immediately whenever possible. (The commands executed can be seen by selecting “Echo Commands” in the “Options” menu of the **“Executive Window”**.) In case some mandatory parameters are missing the corresponding “Command Argument Panel” is displayed, and he remaining arguments have to be filled in. The command is executed then by pressing the “OK” or “Execute” button. (Note that if it is not the last one in the sequence of commands bound to the menu item, the application is blocked until the “OK” or “Cancel” button is pressed.)

All the application specific definitions for the entities accessible through the browser (objects, browsables and action menus) have to be made in the “Command Definition File” (CDF) with a very simple and easy-readable syntax. This will be described in detail in the sections 3.3 and 3.4.

The two lines text/label area at the top displays information about (Fig. 2.11):

- the current path (or directory) for the selected browsable ❶ (entry “Path:”). The directory can be changed by pointing at the tail of the wanted sub-path and clicking the left mouse button. Clicking a second time on the same path segment performs the directory change and updates the “DirList” window with the list of objects.
- the number of objects of all the different classes defined for the selected browsable in the current directory ❷.

The two lines of information at the bottom are filled with (Fig. 2.11):

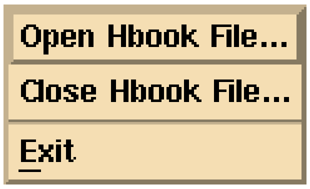
- a short description of the browsable which is currently selected ❺ (entry “File:”),
- a short description of the object which is selected in the “object window” for a given browsable ❻.

Below follows a description of the different Browser menus:

File

The **File** menu can be filled by the application with menu entries (buttons) which give access to the commands that can be used to connect or de-connect a new browsable at run time (e.g. in PAW++ the commands to open or close ZEBRA/RZ files).

These buttons/menu entries are automatically generated from the definition of the action menus for the browsables made in the CDF (see 3.4.3). For example, the **File** menu in the PAW++ **“Main Browser”** is shown below. The last entry of this menu is always “Exit”, to exit from the application.



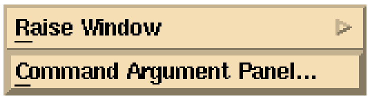
Open Hbook File...	Open one ZEBRA/RZ file.
Close Hbook File...	Close one ZEBRA/RZ file.
Exit	Exit from the application.

View

The **View** menu allows to change the way objects are displayed or selected.

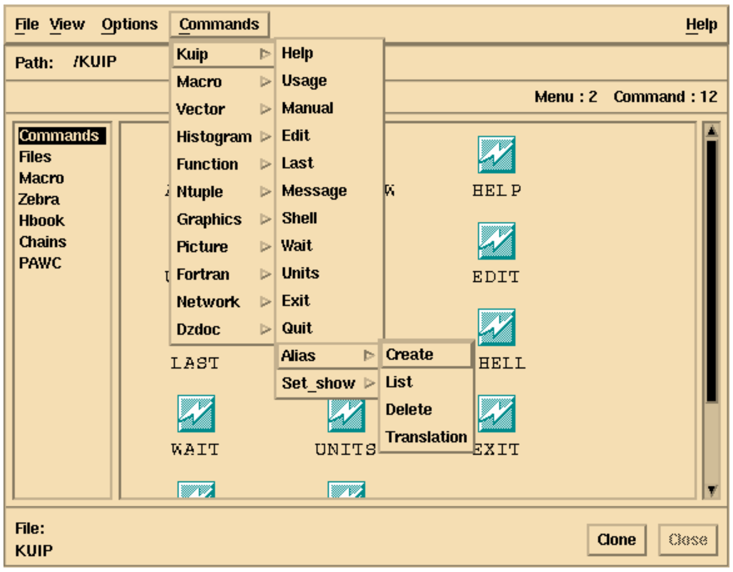
Icons	display objects with normal size icons and names (default).
Small Icons	display objects with small icons and names.
No Icons	display objects without icons, but names and small titles.
Titles	display objects without icons, but long titles.
Select All	select all the objects.
Filter...	ask for a filter to be put on object names.

Options



Raise Window	"cascade button" with the list of all opened windows. Selecting one of this window will pop-up the window on top of the others.
Command Argument Panel	selecting this entry will prompt the user for a command name. If the command is valid then the corresponding "Command Argument Panel" with the list and description of all parameters will be displayed. If the command is ambiguous (e.g. command "list") the user will be proposed a list of all the possible commands. He can then select one and the corresponding "Command Argument Panel" will be displayed. If the command does not exist an error message is displayed.

Commands



This menu gives access to the complete tree of commands defined by KUIP and the application in the form of a pull-down menu. When a terminal item (command) in this menu is selected then the corresponding "Command Argument Panel" is displayed. The functionality of this menu is quite similar to the browsable "Commands" (this is just a matter of taste whether the user prefer to access commands through this pull-down menu or through the "Commands" browser).

Help

On {Appl.}	Help specific to the application (has to be written in the CDF (Command Definition File)).
On {Appl.} Resources	Help specific to the application resources (has to be written in the CDF (Command Definition File)). Resources control the appearance and behavior of an application.
On Kuip Resources	List the X resources available to any KUIP/Motif based application.
On Browser	Help on the KUIP/Motif Browser interface (**"Main Browser"**).
On Panel	Help on the KUIP/Motif "**PANEL** interface".
On System Functions	List KUIP all internal system functions currently available.

2.8.1.2 Browser Setting or Initialization

The following KUIP/Motif command can be used to set up the browser in a given state, without having to click with the mouse:

```
/MOTIF/BROWSER browsable [path]
```

- *browsable* is the name of the file (browsable) you want to open (corresponding item is selected in the list of browsables).
- *path* (optional) is the pathname to be used for this browsable.

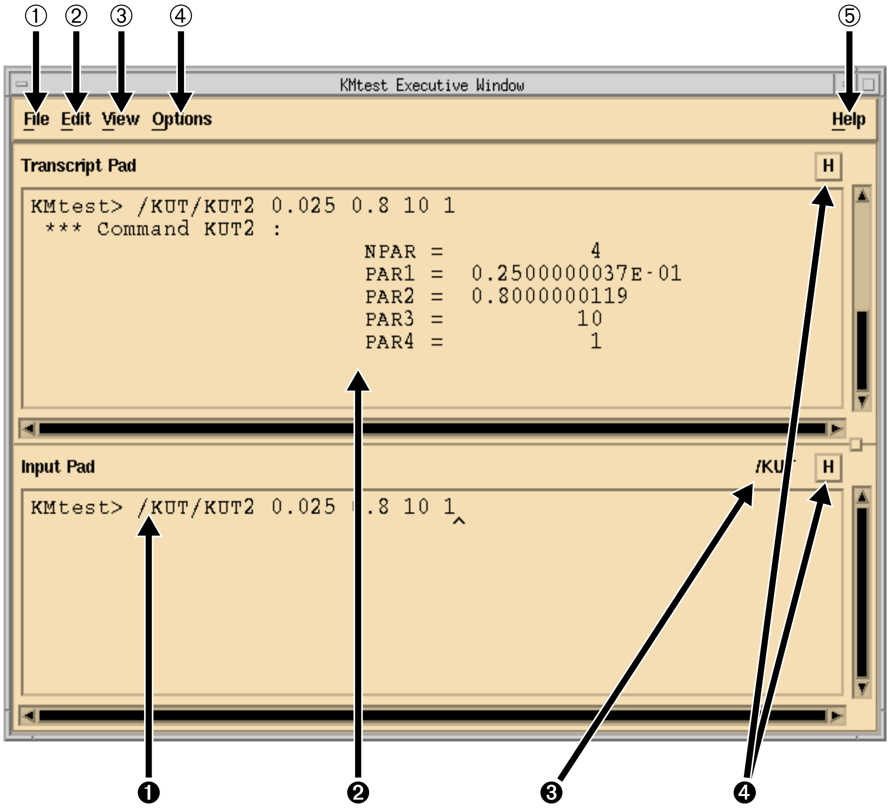
E.g. If you want to open the browser in the state displayed in Fig. 2.11, without having to click with the mouse, you can execute the KUIP command:

```
/MOTIF/BROWSER Files /neutrons/cremel/kuip
```

It is also possible, for the application programmer, to initialize the browser in a certain state when the application is starting. For that we provide the Motif user callable C routine *km_browser_set* (see section 3.6.2) which can be called just before entering the Motif main loop.

2.8.2 KXTERM: the KUIP Terminal Emulator (or "Executive Window")

This terminal emulator combines features from Apollo DM pads (**Input Pad** and **Transcript Pad**, automatic file backup of **Transcript Pad**, string search in pads, etc.) and the Korn shell emacs-style command line editing and command line recall mechanism.



① Menu bar entry "File".
② Menu bar entry "Edit".
③ Menu bar entry "View".
④ Menu bar entry "Options".
⑤ Menu bar entry "Help".

❶ **Input Pad**
❷ **Transcript Pad**
❸ Current working directory indicator.
❹ Hold buttons.

Figure 2.12: KXTERM (KUIP/Motif **"Executive Window"**)

2.8.2.1 Description and Behavior

KXTERM (or what we call the **"Executive Window"** in a KUIP based application) is composed of three main parts (Fig. 2.12):

- A "menu bar" with the menu entries "File" ①, "Edit" ②, "View"③, "Options"④, and "Help"⑤,.
- A **Transcript Pad** ❷ which contains any kind of output coming from KUIP or from the application.
- An **Input Pad** ❶ which is an edit-able "scrolled window" where the user can type commands.

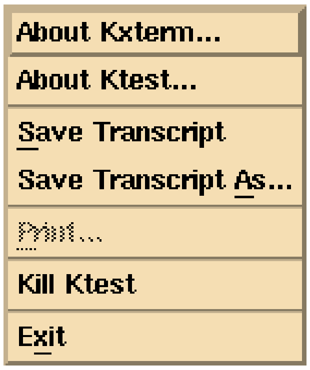
Commands are typed in the input pad behind the application prompt. Via the toggle buttons ❹ labeled "H" the **Input Pad** and/or **Transcript Pad** can be placed in hold mode. In hold mode one can paste or type a number of commands into the **Input Pad** and edit them without sending the commands to the application. Releasing the hold button will causes Kxterm to submit all lines, up to the line containing the cursor, to the application. To submit the lines below the cursor, just move the cursor down. In this way one can still edit the lines just before they are being submitted to the application.

Commands can be edited in the **Input Pad** using emacs-like key sequences (see section 2.8.2.2). The **Transcript Pad** shows the executed commands and command output. When in hold mode the **Transcript Pad** does not scroll to make the new text visible.

Every time the current directory is changed, the **Current working directory indicator** ❸ is updated. The current working directory is the one which is currently selected in the **"Main Browser"**.

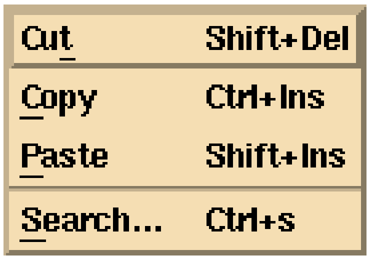
Below follows a description of the different Kxterm menus. All Kxterm menus can be dynamically extended by the application (see section 3.4.5).

File



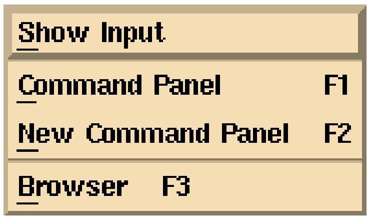
About Kxterm...	Displays version information about Kxterm.
About {Appl.} ...	Displays version information about the application Kxterm is servicing.
Save Transcript	Write the contents of the **Transcript Pad** to the current file. If there is no current file a file selection box will appear.
Save Transcript As...	Write the contents of the **Transcript Pad** to a user-specified file.
Print...	Print the contents of the **Transcript Pad** (not yet implemented).
Kill {Appl.}	Send a SIGINT signal to the application to cause it to core dump. This is useful when the application is hanging or blocked. Use only in emergency situations.
Exit	Exit Kxterm and the application.

Edit



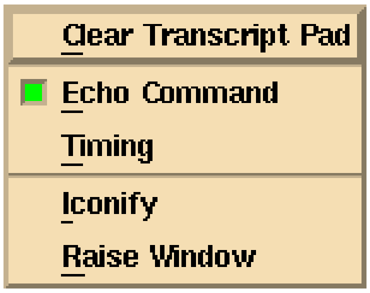
Cut	Remove the selected text. The selected text is written to the Cut & Paste buffer. Using the “Paste” function, it can be written to any X11 program. In the **Transcript Pad** “Cut” defaults to the “Copy” function.
Copy	Copy the selected text. The selected text is written to the Cut & Paste buffer. Using the “Paste” function, it can be written to any X11 program.
Paste	Insert text from the Cut & Paste buffer at the cursor location into the **Input Pad**.
Search...	Search for a text string in the **Transcript Pad**.

View



Show Input	Show in a window all commands entered via the **Input Pad**.
Command Panel	Gives access to the KUIP/Motif “**PANEL** interface” for a panel which has been predefined in a KUIP macro file (see section 2.8.3).
New Command Panel	Gives access to the KUIP/Motif “**PANEL** interface” for setting a new and empty panel to be filled interactively (see section 2.8.3).
Browser	Display another instance of the browser.

Options



Clear Transcript Pad	Clear all text off of the top of the **Transcript Pad**.
Echo Command	Echo executed commands in **Transcript Pad**.
Timing	Report command execution time (real and CPU time).
Iconify	Iconify Kxterm and all windows of the application.
Raise Window	Display a list of all windows connected to the application. The user can select the window he wants to pop-up.

Help

On Kxterm	The help you are currently reading.
On Edit Keys	Help on the emacs-style edit key sequences.
On {Appl.}	Help specific to the application (has to be written in the CDF).
On {Appl.} Resources	Help specific to the application resources (has to be written in the CDF). Resources control the appearance and behavior of an application.
On Kuip Resources	List the X resources available to any KUIP/Motif based application.
On Browser	Help on the KUIP/Motif Browser interface (**"Main Browser"**).
On Panel	Help on the KUIP/Motif "**PANEL** interface".
On System Functions	List all KUIP internal system functions currently available.

2.8.2.2 Edit Key Sequences

Please note that "C-b" means holding down the Control key and pressing the "b"-key. "M-" stands for the Meta or Alt key.

```
C-b:            backward character
M-b:            backward word
Shift M-b:      backward word, extend selection
M-[:            backward paragraph
Shift M-[:      backward paragraph, extend selection
M-<:            beginning of file
C-a:            beginning of line
Shift C-a:      beginning of line, extend selection
C-osfInsert:    copy to clipboard
```

```
Shift osfDelete:  cut to clipboard
Shift osfInsert:  paste from clipboard
Alt->:            end of file
M->:              end of file
C-e:              end of line
Shift C-e:        end of line, extend selection
C-f:              forward character
M-]:              forward paragraph
Shift M-]:        forward paragraph, extend selection
C-M-f:            forward word
C-d:              kill next character
M-BS:             kill previous word
C-w:              kill region
C-y:              yank back last thing killed
C-k:              kill to end of line
C-u:              kill line
M-DEL:            kill to start of line
C-o:              newline and backup
C-j:              newline and indent
C-n:              get next command, in hold mode: next line
C-osfLeft:        page left
C-osfRight:       page right
C-p:              get previous command, in hold mode: previous line
C-g:              process cancel
C-l:              redraw display
C-osfDown:        next page
C-osfUp:          previous page
C-SPC:            set mark here
C-c:              send kill signal to application
C-h:              toggle hold button of pad containing input focus
F8:               re-execute last executed command
Shift F8:         put last executed command in input pad
Shift-TAB:        change input focus
```

2.8.3 User Definable Panels of Commands ("PANEL interface")

KUIP/Motif includes a built-in "**PANEL** interface" that allows to define command sequences which are executed when the corresponding button in the panel is pressed.

As you will see, this "**PANEL** interface" is quite powerful compared to the "STYLE GP" which was available in the basic KUIP for graphical screens. In particular it is possible to set-up panels with graphical keys (icons) representation.


```
NEWPANEL 4 6 'First panel' _
        250 200 500 600
```

This KUIP/Motif command creates an empty panel with 4 rows and 6 columns of buttons. The title of this panel will be set to "Gtest First panel" ("Gtest" is the application class-name). The panel size in pixels is 250 (width) x 200 (height), and the panel position (in pixels) is 500 (along X axis), 600 (along Y axis).

Figure 2.13: New Panel of Commands

2.8.3.1 New Panel

It is possible to fill a new and empty panel interactively (see section 2.8.3.4) giving a label to each button.

In the top menu bar 3 pull-down menus ('File", "View" and "Help") are available. The pull-down menu "File", whose contents is displayed, contains the 2 items "Save" (to save the actual panel configuration after editing) and "Close" (to close the panel and erase it from the screen). The "View" menu contains various options for displaying the same panel in different ways (see section 2.8.3.3), and the "Help" menu contains various items to help the user concerning this panel interface.

This new panel definition can also be done with the command PANEL using the sequence

```
PANEL 0
PANEL 4.06 ' '
PANEL 0 D 'This is my first panel' 250x200+500+600
```

You can get automatically access to the command "NEWPANEL" (and its corresponding "Command Argument Panel") by selecting the menu item "New Command Panel" in the "View" menu of the **"Executive Window"** (KXTERM, Fig. 2.8.2.1).

2.8.3.2 Predefined Panel of Commands

The command "PANEL" for a key (or button) definition has to be used if you want to describe your panel in a KUIP macro file in order to keep trace of the panel definition, and be able to retrieve it later on. You can predefine as many panels as you want, and you can easily access them by selecting the menu item "Command Panel" in the "View" menu of the **"Executive Window"** (section 2.8.2.1).

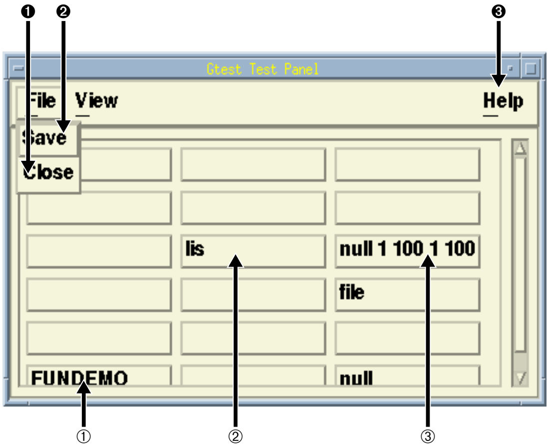
You have to describe in the KUIP macro file(s) each button individually. You can also request the macro(s) execution in your "KUIP logon" file so that the panel(s) will be automatically displayed at the beginning of the session.

The general syntax of the KUIP/Motif command "PANEL" for a key definition is:

```
panel x.y command [label] [pixmap]
```

– *x.y* is the key position (column and row number),
– *command* is the complete command (or list of commands) to be executed when the corresponding button is pressed,
– *label* (optional) is an alias-name for this command. If specified, this alias-name is used for the button label (when the appropriate "View" option is selected) instead of the complete command (which is generally too long for a "user-friendly" button label).
– *pixmap* (optional) has to be specified for graphical keys (fully described in the next section 2.8.3.3).

An example of a panel definition is given in figure 2.14.



KUIP Macro for panel definition

```
*
* MOTIF_PANEL panel_test.kumac
*
panel 0
panel 3.02 'list'
panel 3.03 'null 1 100 1 100'
panel 4.03 'file'
panel 6.01 'FUNDEMO'
panel 6.03 'null'
panel 0 d 'Test Panel' 450x250+600+600
```

❶ Close button (to close panel)
❷ Save button (to save panel into a KUIP macro file)
❸ Access to various "helps" on the "**PANEL** interface"

① ② ③ User defined buttons

Figure 2.14: Predefined Panel of Commands

2.8.3.3 Panel with Graphical keys (Icons) and "View" Selection

As seen in the previous section, the general syntax of the KUIP/Motif command "PANEL" for a key definition allows the user to define graphical keys (or buttons) where pixmaps are used instead of alpha-numerical labels:

```
panel x.y command [label] [pixmap]
```

The last parameter *pixmap* (optional) is the pixmap to be used for representing the key (button) graphically. If it is specified the graphical representation is displayed by default. It is anyway always possible at run time to ask for an alpha-numerical representation by selecting the appropriate entry in the "View" menu of the panel.

The application can define its own icons (pixmaps). This can be done by the application programmer in the CDF (following the KUIP/Motif directive "`>Icon_bitmaps`", described in section 3.4.5.3) or by the user himself (at run-time and for his own user-defined panels of commands) using the KUIP/Motif command:

```
/MOTIF/ICON pixmap filename
```

- *pixmap* is the name given to the icon bitmap and used in the "panel" command for a graphical key definition.
- *filename* is the name of the file where the icon bitmap data is stored.

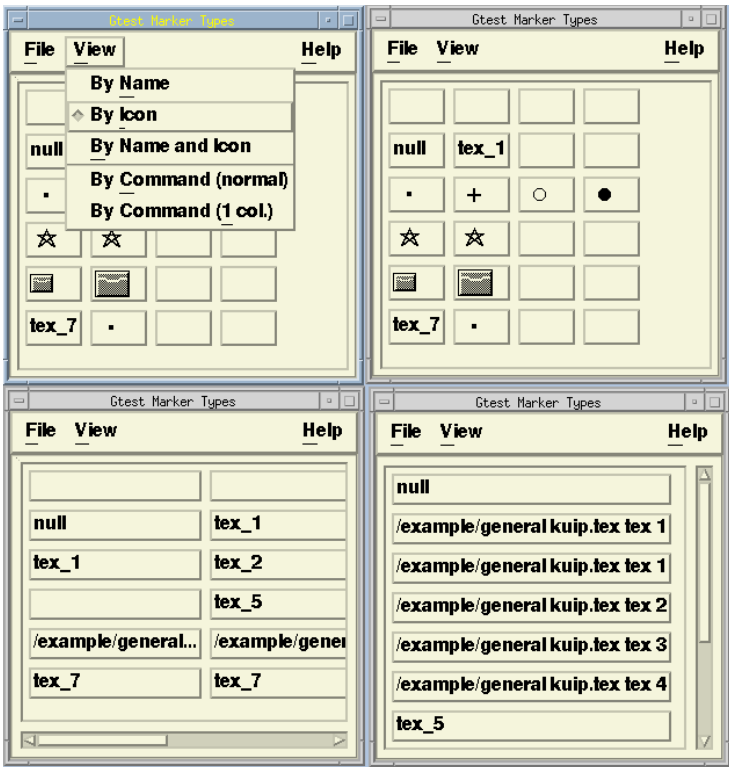
N.B. All application-defined pixmaps (in the CDF) are available to the user in the "panel" command, without having to use this "/MOTIF/ICON" command. This command is only useful when you want to make new icons known by the application (and the command "panel").

To create a new icon bitmap (or pixmap) one can use the X11 standard bitmap editor "bitmap". E.g., to get a 20×20 pixel icon called "`m1`", one can type: `bitmap m1.bm 20x20`. The output file `m1.bm` containing "`#define m1_width 20` ..." has to be referred in the command "/MOTIF/ICON" (with the correct path for the filename), e.g. `/MOTIF/ICON m1 /user/.../.../m1.bm`

The following KUIP macro is a general example for a panel definition with graphical keys.

```
****************************************************
*                                                  *
*             *** panel.kumac ***                  *
*                                                  *
* General example for a panel with icons definition *
*                                                  *
*                                                  *
****************************************************
*
* Icon bitmaps
*
/motif/icon m1 mk1.bm
/motif/icon m2 mk2.bm
/motif/icon m3 mk3.bm
/motif/icon m4 mk4.bm
/motif/icon m5 mk5.bm
*
* Panel keys definition
* N.B. General syntax:
*      panel r.c command [label] [pixmap]
*      label --> command alias
*                (written in the panel and executed for <Button press>).
*                if <label> (optional) is defined then:
*                   /KUIP/ALIAS/CREATE <label> <command>
*                is automatically generated.
*                if <label> is not defined then "command" is used
*                for button label.
*
panel 0
panel 2.01 null
panel 2.02 tex_1
panel 3.01 '/example/general kuip.tex tex 1' 'tex_1' m1
```

```
panel 3.02 '/example/general kuip.tex tex 2' 'tex_2' m2
panel 3.03 '/example/general kuip.tex tex 3' . m3
panel 3.04 '/example/general kuip.tex tex 4' . m4
panel 4.01 ' ' . m5
panel 4.02 'tex_5' . m5
panel 5.01 '/example/general kuip.tex tex 6' . sm_menu
panel 5.02 '/example/general kuip.tex tex 6' . big_menu
panel 6.01 '/example/general kuip.tex tex 7' 'tex_7'
panel 6.02 '/example/general kuip.tex tex 7' 'tex_7' m1
panel 0 d 'Marker Types' 300x300+500+500
```



''By Name'' (bottom left): The panel is displayed with alphanumeric labels. If the alias-name "label" is specified in the "panel" command it is used for the button label, otherwise the complete command is displayed.

''By Icon'' (top right): The panel is displayed with graphical labels (icons), if "pixmap" is specified in the "panel" command. Otherwise "label" or the complete command are used instead (no graphical representation). This "view" setting is the default one (the setting can be changed interactively at run time, and the default setting can be changed with the appropriate resource in the ".Xdefaults" for each user individually).

''By Name and Icon'': The panel is displayed with both alphanumeric and graphical (if any) labels. (Not yet implemented ...).

''By Command (normal)'' The panel is displayed with the complete command names. The arrangement of the buttons stay the same (which might not be very convenient ... See below).

''By Command (1 col.) (bottom right): The panel is displayed with the complete command names BUT the arrangement of the buttons is modified: all buttons are displayed on one column, and "blank" buttons are suppressed (this can save a lot of space, and is more user-friendly, for this kind of viewing option).

Figure 2.15: Panel "View" Selection

Figure 2.15 shows the panel defined in the macro listed above with different "View(ing)" options. In the first window (top/right) the "View" menu is displayed, with the different possibilities which are offered to the user to see the same panel in different ways.

2.8.3.4 Panel Edition and Saving

All the panels (new or predefined) can be edited interactively. Clicking with the left mouse button on a panel button removes its definition. Clicking with the right mouse button on an empty panel button the user will be asked to give a definition to this button (figure 2.16).

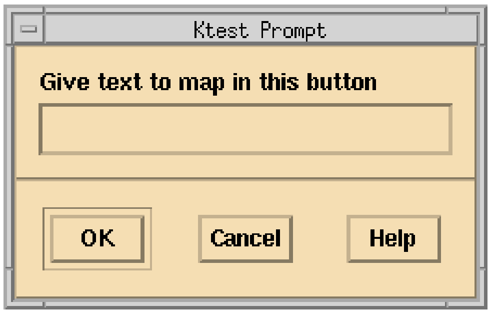


Figure 2.16: Interactive panel button definition

The PANEL commands needed to recreate a panel can be automatically saved into a macro file by pressing the "Save" button ❷ (Fig. 2.14). The panel configuration with its current size and position (which can be modified interactively) is kept into the macro. Panels can be reloaded either by executing the command 'PANEL 0 D' or by pressing the "Command Panel" button in the "View" menu of the **"Executive Window"** and entering the corresponding macro file name.

Some characters in the panel keys/buttons have a special meaning:

– The dollar sign inside a key is replaced by additional keyboard input. For example:

```
'V/PRINT V($)'    | entering 11:20 will execute V/PRINT V(11:20)
```

– Keys ending with a double minus sign make an additional request of keyboard input. For example:

```
V/PRINT V--'     | entering AB will execute V/PRINT VAB
```

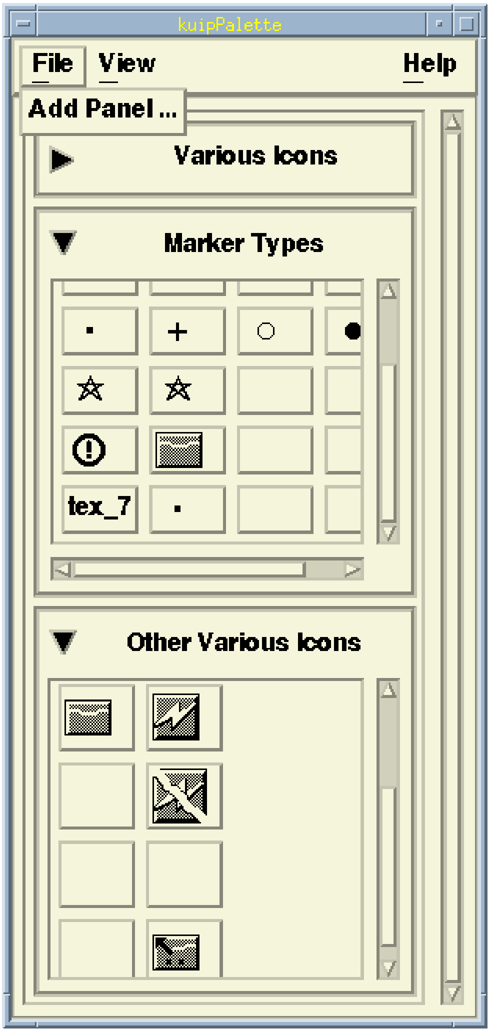
2.8.3.5 "Multi_panel" or Palette of panels Definition

It may be nice or more user-friendly to group a certain number of panels (related to similar actions or objects to be manipulated) in a so-called "palette" of panels. This is possible with the KUIP/Motif command "MULTI_PANEL" which opens such a widget. [5]

```
/MOTIF/MULTI_PANEL [title] [geometry]
```

E.g. MULTI_PANEL 'My Palette' '200x100+0+0' will display a "multi_panel" widget with title "My Palette" and geometry "200x100+0+0" (Position=0,0 in X and Y, width=200, height=100). When this command is executed all panel definitions and executions will go into this "multi_panel" (or palette)

[5] For those who are familiar with the "UIMX" User Interface Management System, this is an emulation of the "Palette" widget which is built-in inside this program.



User-defined KUIP/Motif “palette” with 3 panels :

“Various Icons” : this panel is not displayed (arrow turned left to right) at the moment. One would just have to press the arrow button to make it visible ...

“Marker Types” : this user-defined panel is visible (arrow turned top to down). One can turned it off by pressing the arrow button.

”Other Various Icons” : this user-defined panel is also visible.

Figure 2.17: Multi_panel (or Palette)

widget. This can be done simply by executing KUIP macro(s) containing your panel definition(s), or by selecting the "Add button" entry in the menu “File” available in the "multi_panel" widget. To terminate a "multi-panel" setting one just have to type: `MULTI_PANEL end`. This means that the following panel definitions and executions will be displayed as individual panels and will not go into this "palette" anymore, unless another palette is opened (by executing again the command “MULTI_PANEL”). Then the panels will go into that new palette.

The following sequence of commands (which can be put inside a macro) can be used to set up a palette:

```
MULTI_PANEL
EXEC PANEL1.KUMAC
EXEC PANEL2.KUMAC
EXEC PANEL3.KUMAC
MULTI_PANEL end
```

N.B. `panel1.kumac`, `panel2.kumac`, and `panel3.kumac` are KUIP macro files with “usual” panel setting and definition.

Figure 2.17 shows an example of a user-defined palette (with some predefined panels). The "arrow buttons" can be pressed either to reduce the panel to a label containing the panel title (arrow button is then turned left to right) or to display it (arrow button turned up to down). One can see that the KUIP/Motif "palette" is a good way to have many panels defined and save space on the screen.

2.8.4 KUIP/Motif X-Windows Resources

X-Windows resources control the appearance and behavior of an application. Users who are not pleased with the default values proposed for any resources that can affect their KUIP/Motif based application, can override them by specifying their own values in the standard X11 way : i.e. by editing their private ".Xdefaults" file or the system wide "/usr/lib/X11/app-defaults/<appl_class>".

Each new resource has to be specified on a separate line. The syntax for editing one specific resource is always the following:

<appl._class>*<resource_name>: <resource_value>

where:

- "appl._class" has to be replaced by the real application class name (e.g. "Paw++" for PAW++) which is the input parameter of the routine `KUWHAM` (section 4.1.2).
- "resource_value" is the value to be given to the corresponding "resource_name". It can be an integer, a boolean value, a color, a font, or any kind of predefined syntax (e.g. for geometry).

The following is a (non exhaustive) list of the most important or frequently used X-Windows resources for a KUIP/Motif based application. The default values provided by KUIP/Motif (if any) are put inside "[]".

- Background and foreground color for all windows (except KXTERM):
  ...*background: ...
  ...*foreground: ...
- Geometry ([width]x[height]+[xpos]+[ypos]) of the **"Executive Window"** (KXTERM):
  ...*kxtermGeometry: ... [650x450+0+0]
- Geometry of the Browser(s):
  ...*kuipBrowser_shell.geometry: ... [-0+0] (1) or [+0+485] (2)
  (1) without any graphics window - (2) with graphics window(s) managed by HIGZ.
- Geometry of the Graphics Window(s) (if any):
  ...*kuipGraphics_shell.geometry: ... [600x600-0+0]
- Character font for menus, buttons and dialog area:
  ...*fontList: ... [-adobe-helvetica-bold-r-normal--12-120-75-75-p-70-iso8859-1]
- Character font for the **Input Pad** and **Transcript Pad** (KXTERM):
  ...*kxtermFont: ... [*-courier-medium-r-normal*-120-*]
- Character font for the "HELP" windows:
  ...*helpFont: ... [*-courier-bold-r-normal*-120-*]

- Character font for all "Text" widgets:

  ...*XmText*fontList: ...

  ...*XmTextField*fontList: ...
- Character font for the icon labels in the browser(s) **"Object window"**:

  ...*dirlist*fontList: ...
- Background and foreground colors for the **"Object window"** in browser(s):

  ...*dirlist*background: ...

  ...*dirlist*foreground: ...
- Background and foreground colors for the icons associated to the object class "objclass":

  ...*dirlist*<objclass>*iconBackground: ... [white]

  ...*dirlist*<objclass>*iconForeground: ... [black]
- Background and foreground colors for the icon-labels associated to the object class "objclass":

  ...*dirlist*<objclass>*iconLabelBackground: ... [white]

  ...*dirlist*<objclass>*iconLabelForeground: ... [black]
- Possibility to turn on/off the zooming effect when traversing directories structures inside the browser(s):

  ...*zoomEffect: ... [on]
- Speed of the zooming effect in the browser(s) when turned on:

  ...*zoomSpeed: ... [10]
- Double click interval in milliseconds (time span within which 2 button clicks must occur to be considered as a double click rather than two single clicks):

  ...*doubleClickInterval: ... [250]
- Background and foreground colors for the **"Browsable window"** in browser(s):

  ...*fileList*background: ...

  ...*fileList*foreground: ...
- Focus policy:

  ...*keyboardFocusPolicy: ...

  If "explicit" focus is set by the mouse or a keyboard command. If "pointer" focus is determined by the mouse pointer position.

The appearance and behavior of the **"Executive Window"** are managed by "KXTERM" whose classname is "KXterm". It means that, for instance, to change the background and foreground color of the **"Executive Window"**, one has to override the following resources:

KXterm*background: ...

KXterm*foreground: ...

To this list of resources one can add all the resources which can affect any Motif widgets which are used by KUIP/Motif.

Concerning the appareance of the icons built-in inside KUIP/Motif (browsers for "Commands", "Files" and "Macro"), the classes of objects which are currently predefined are:

```
Cmd        -- Command
InvCmd     -- Deactivated command
```

```
Menu       -- Menu tree
MacFile    -- Macro File
RwFile     -- Read-write file
RoFile     -- Readonly file
NoFile     -- No access file
ExFile     -- Executable file
DirFile    -- Directory
DirUpFile  -- Up directory (..)
```

2.9 Nitty-Gritty

2.9.1 System dependencies

KUIP tries to provide as far as possible a homogenous environment across different operating systems and hardware platforms. Here we want to summarize the remaining system-dependencies. To a large extend the comments made on Unix apply also to the MS-DOS and Windows/NT implementations.

2.9.1.1 SHELL command

The `SHELL` command allows to pass a command line to the underlying operating system for execution. If used without arguments the `SHELL` command suspends the application program and allows to enter OS commands interactively. When leaving the subprocess, either with the command `return` or `exit` depending on the system, the application resumes execution.

Unix The command `HOST_SHELL` defines the shell to be invoked. The start-up value is taken from the environment variable `SHELL` or set to an appropriate default such as `/bin/sh`. On some Unix implementations the `SHELL` command can fail if there is not enough free swap space to duplicate the current process.

VMS The `SHELL` command spawns a subprocess with a DCL command processor. This is notoriously slow and there is no way to combine several DCL commands into one `SHELL` command.

VM/CMS "`SHELL cmd`" first tries to find the file "`CMD EXEC *`" and execute it as a REXX script. Otherwise the command is passed as-is which will either run "`CMD MODULE *`" or execute the genuine CMS command `CMD`. There are some restrictions on the kind of modules that can be executed in CMS subset mode. CP commands have to be prefixed, e.g. "`SHELL CP Q TIME`".

2.9.1.2 EDIT command

The `EDIT` command allows to edit a file without leaving the application program. The command `HOST_EDITOR` defines the editor to be invoked. The start-up value is taken from the environment variables `KUIPEDITOR`, `EDITOR`, or set to a system dependent default.

`HOST_EDITOR` sets the shell command (sans filename) for starting the editor. Some values have a system dependent special meaning.

Unix The default editor is `vi`. The shell command containing a "`&`" does not necessarily mean that the editor will run as a background process (see section 2.9.2).
On Apollo/DomainOS "`DM`" uses the Display Manager editor. This is the default if the application program is started from a DM pad.

VMS The special names `EDT` and `TPU` use the callable interface to these two editors. The startup time is much less than, for example `EDIT/TPU` which spawns a subprocess. However, there is a problem with the callable `EDT`. If any error condition occurs (invalid filename etc.) the callable `EDT` will be unusable for the rest of the session.

VM/CMS There is only one possible `HOST_EDITOR`: `XEDIT`. For editing large files the virtual machine's size must be dimensioned that the application program and `XEDIT` fit into the available memory at the same time.

2.9.1.3 Exception handling

KUIP installs a signal handler in order to catch exceptions and return to the command input prompt. The command "`BREAK OFF`" disables the signal handler, i.e. the program aborts in case of an exceptions. For some systems "`BREAK ON`" allows to request a traceback of where the exception has happened.

There are two major types of exceptions caught by the signal handler. Program exceptions indicate either a bug in the application program (or KUIP) or insufficient protection against invalid input:

Floating point exceptions are caused by divide by zero, floating point overflow, square root of negative numbers etc. Floating point underflows are usually silently ignored and the result is treated as being zero.

Segmentation violation indicates an attempt to read or write a memory location outside the address space reserved by the process, e.g. if an array index is out of bounds. In C code it is most often caused by dereferencing a `NULL` pointer which is prohibited on many systems.

Bus error is usually caused by an unaligned access. Most RISC processors have strict requirements for properly aligned data.

Illegal instruction can mean that the program tries to executed data as code, for example if the return address on the stack has been overwritten.

Don't be surprised if the program shows irregular behaviour after an exception!

The second type of exceptions handled by the KUIP signal handler are user breaks. Hitting the break key (usually `Ctrl`-C) aborts a running command and returns to the input prompt. Using `Ctrl`-C is potentially unsafe unless the application is properly coded to block keyboard interrupts in critical sections. Otherwise the interrupt can happen at an inconvinient moment which leaves the program's data structures in an inconsistent state. The signal handler prompts the user after three consecutive keyboard interrupts to allow exiting from a run-away process.

Unix The actual break key can be changed with the Unix command `stty`. The default setup usually is "`stty intr ^C`". Unix provides a second kind of keyboard interrupt which is intentionally **not** caught by the KUIP signal handler to allow killing run-away processes. A convenient setting is "`stty quit '\\'`"
User break interception does not work for Windows/NT. Tell Microsoft that signal handlers are pretty useless if they are not allowed to use `printf` and `longjmp`.

VMS The user break key is `Ctrl`-C. `Ctrl`-Y is treated like `Ctrl`-C, i.e. it does not bring up the DCL prompt.

VM/CMS There is no user break for VM/CMS. To abort a run-away session use

```
#CP EXT
HX
```

2.9.2 The edit server

By default editing from within a KUIP application is synchronous, i.e. the application is suspended until the editor terminates. On a workstation this is an inconvenient restriction because the editor can run in a separate window while the application continues to accept commands.

Although not an issue for the `KUIP/EDIT` command itself there are applications (notably CMZ) which have to process the file content after it has been edited. Therefore the editor cannot be simply started as a background process.

To take care of this problem KUIP provides a facility called the "edit server". Instead of calling the editor directly, KUIP starts the editor server as a background process which leaves the application program ready to accept more commands. The server invokes the editor and waits for it. When the editor terminates the server informs the application program about the file which is ready. KUIP can then call the application routine for processing the edited file.

The processing routine cannot be called at the very instant the file is ready. KUIP waits until the user hits the `RETURN`-key to execute the next command. The file is then checked in *before* the command just entered is executed.

As a protection especially for users working alternately on a terminal or on a workstation KUIP does not try asynchronous editing if one of the following conditions is missing:

- The edit server module `kuesvr` must be found in the search path.
- The editor command set by `HOST_EDITOR` must end with an ampersand ("`&`").
- The environment variable `DISPLAY` must be set.

Note that the editor command must create its own window, possibly by wrapping the editor into a terminal window. For convenience "`HOST_EDITOR 'vi &'`" is interpreted automatically as "`xterm -e` ***cmd*** `&`".

The edit server cannot be used for the Apollo DM editor. Some Unix windowing editors tend to fork themselves as a detached process by default. For example the `jot` editor found on Silicon Graphics systems requires a special option "`-noFork`". Otherwise the edit server and the application think that the editor has already terminated leaving the file unchanged.

In the KUIP/Motif interface it is essential to use the edit server mechanism. Otherwise invoking the editor from a pop-up menu freezes the screen when the right-hand mouse button is pressed before the subprocess terminates[6]. The screen can only be unlocked by logging in remotely and killing the application program.

For asynchronous editing on VMS either the Motif version of TPU must be used or the hosteditor command must create its own terminal window, e.g.

```
HOST_EDITOR TPU/DISPLAY=MOTIF
HOST_EDITOR 'CREATE/TERM/WAIT EDT'
```

[6] Can somebody elucidate this problem or knows a workaround? It seems that the application does not receive the button-release event and therefore the Motif pop-up menu never releases the pointer grab???

2.9.3 Implementation details

2.9.3.1 Command search order

With the various possibilities of changing the interpretation of a command line it is sometimes important to know the exact order in which the different mechanisms are applied:

1 If the input line contains a semicolon line separator (section ??), split off the front part and deal with the rest later. In case the line separator is "`;&`" or "`;!`" the execution of the remaining line depends on the status code of the first command.

2 If executing a macro script, substitute all variables by their values.

3 If the first token is a command alias (section ??), substitute it by its value. If the replacement contains a semicolon line separator, start again at step 1. In order to protect against recursive aliases stop if a reasonable upper limit on the number of iterations is exceeded.

4 Unless the command name belongs to the `KUIP/ALIAS` menu, substitute argument aliases. Argument aliases can occur in the command name position but they may not contain semicolon line separators.

5 Substitute system functions (section ??).

6 If executing a macro with "`TRACE ON`", show the present command line. If "`TRACE ON WAIT`" prompt for further actions:

- execute command
- skip execution of this command
- quit execution of macro script
- continue macro execution without further prompting

7 Separator first token (command name `cmd`) from the rest of the line (argument list).

8 Unless executing a macro, if "`DEFAULT -AutoReverse`" (section ??) is active and `cmd.kumac` is found in the macro search path, transform the command name into `EXEC`. The command token itself has to be put back in the front of the other argument. If the command token contains a "`#`" character we had to separate the front part before searching for the `.kumac` file.

9 Match `cmd` as abbreviation against the command tree:

- If `cmd` begins with a slash, start at the top menu.
- Otherwise start at the `SET/ROOT` menu; if there is no match and the current root is not the top menu itself, start again at the top menu.

10 Unless executing a macro, if "`DEFAULT -Auto`" is active and `cmd` is either not a command or ambiguous, try again procedure of step 8.

11 If a `SET/COMMAND` template is defined and `cmd` is unknown as command name, i.e. not just ambiguous, apply the template replacement and go back to step 1. `SET/COMMAND` must be disabled temporarily to avoid an infinite recursion in case the template itself is an invalid command.

12 If `cmd` is ambiguous, show the list of possible solutions and exit.

13 If `cmd` is not a valid command name, print error message and exit.

14 Otherwise tokenize the argument list and call the action routine for the command.

2.9.3.2 Name spaces

There is an admittedly confusing difference in the characters allowed to form the various KUIP identifiers which we summarize here:

Alias names allow letters, digits, "_", "@", "-", "$".
Macro variable names allow letters, digits, "_". The first character may not be a digit.
System function names allow letters, digits, "_". The first character may not be a digit. Uppercase and lowercase letters are distinct when the name is looked up as environment variable.
Vector names allow letters, digits, "_", "?". The first character may not be a digit. Names starting with "?" are reserved.

Although not in the hands of the application user but only the application writer:

Command and menu names allow letters, digits, and "_".
Parameter names allow letters, digits, and "_". The first character may not be a digit.

3 Programmer interface

The key to building a user interface with KUIP is the Command Definition File (CDF) which the application write has to provide. As the name already indicates the CDF defines the available commands and their grouping into menus. The CDF directives for the basic KUIP command line interface are described in section 3.2.

Section 3.4 introduces the main concepts of KUIP/Motif and describes the CDF directives specific to this interface, e.g. how to add new object types to the browser.

The CDF is a text file which has to be passed through the KUIP compiler (KUIPC). The output of KUIPC is either Fortran or C source code for a subroutine which the application program has to call at initialization time in order to store the CDF information in memory. Section 3.5 explains how to run KUIPC and the pros and cons of generating Fortran vs. C source code.

Figure 3.1 shows the steps from the CDF to the running application.

3.1 The Command Definition File

The CDF format is line-oriented. Completely blank lines are ignored. An underscore character at the end of a line acts as continuation symbol, i.e. before processing the following line is concatenated to the current line removing the underscore itself.

Directives starts with a "`>`" character in the first column immediately followed by the directive name. Most directives take arguments following on the same line and some directives expect additional information on separate lines. CDF argument lines are tokenized according to the rules:

- Individual arguments must be separated by one or more blanks.
- Strings containing blanks must be delimited by single quote characters, e.g. `'Hello world'`.
- Single quote characters inside a string must be duplicated, e.g. `'N''th value'`.
- A single "`.`" is the place-holder for a null argument.

Names for menus, commands, and parameters are case-insensitive and may consists of letters, digits, minus signs and underscores. For obvious reasons names containing only digits, starting with a minus sign, or ending in an underscore should be avoided. Names may start with a digit.

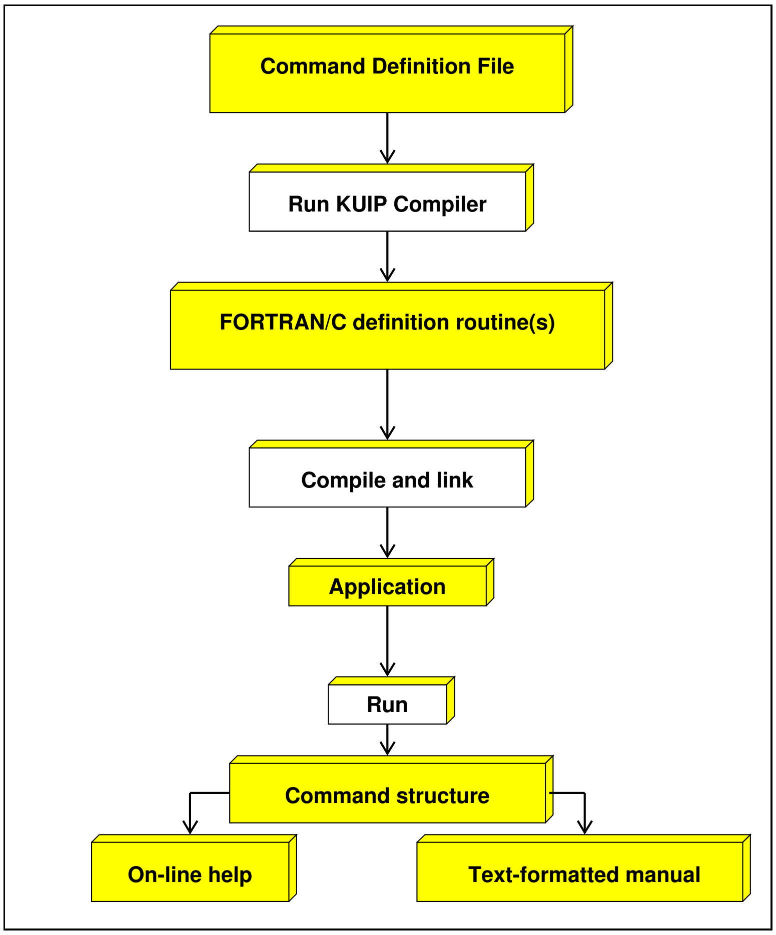


Figure 3.1: Building an application starting from the CDF.

3.2 CDF directives for command definitions

The CDF has to define the command tree structure which can contain three types of elements:

- Menus are the branches of the tree. Each menu should provide a help text to give general information about the commands and sub-menus linked to it.
- Commands are the terminal leaves of the tree. The CDF should contain the help text and define the parameters (if any). Each command definition must specify the action routine which retrieves the command arguments and performs the intended operations.
- Help items are like commands without action routine. Their sole purpose is to provide additional help not related to a specific command or menu. Trying to execute a help item is equivalent to asking `HELP` on that item.

```
>* comment
   ...
```

```
*  comment
```

The `>*` directive starts a block comment. Everything up to the next CDF directive is ignored. A single line comment is introduced by a "`*`" character in the first column.

```
>NAME  definition-routine
```

The `>Name` directive defines the name of the routine which KUIPC will generate, i.e. *definition-routine* must be an acceptable Fortran `SUBROUTINE` name. The application has to `CALL` *definition-name* at initialization time in order to store the command definitions in internal memory.

The first non-comment line in the CDF must always be a `>Name` directive. The CDF may contain several `>Name` directives.

```
>MENU  menu-path
```

Commands are linked into the current menu and the `>Menu` directive allows to changes the menu path. The *menu-path* can consist of several components separated by "`/`". The path is relative to the previously valid path unless *menu-path* starts with a "`/`". The pseudo-name "`..`" refers to the parent menu.

The names for already existing menus may not be abbreviated. If a menu is not already existing it is created and should be followed by a `>Guidance` directive.

The `>Name` directive resets the path to the root menu "`/`". Commands and help item should not be linked directly to the root menu. Therefore the `>Name` directive should always be a followed by a `>Menu` directive.

```
>COMMAND  command-name
```

The `>Command` directive appends a new command to the current menu. The *command-name* must be a bare name without menu path.

The `>Command` directive must be followed by `>Guidance` and `>Action` directives and can be followed by `>Parameters` and `>User_help` directives in any order.

```
>HELP_ITEM  help-item-name
```

The `>Help_item` directive appends a new help item to the current menu. The *help-item-name* must be a bare name without menu path. The *help-item-name* cannot contain a menu path.

The `>Help_item` directive must be followed by a `>Guidance` directive and can be followed by a `>User_help` directive in any order.

Example for the creation of menus

```
>Name KUIPDF
>Menu KUIP
>Guidance
      ...
>Menu ALIAS
>Guidance
      ...
>Menu ../SET_SHOW
>Guidance
      ...
>Menu /MACRO
>Guidance
      ...
>Name VECDEF
>Menu VECTOR
>Guidance
      ...
```

In this example `CALL KUIPDF` creates the menus `/KUIP`, `/KUIP/ALIAS`, `/KUIP/SET_SHOW`, and `/MACRO` and `CALL VECDEF` creates the menu `/VECTOR`.

Example for the creation of commands

```
>Menu /KUIP
>Command HELP
      ...
>Menu ALIAS
>Command CREATE
      ...
>Command DELETE
      ...
```

In this example the commands `/KUIP/HELP`, `/KUIP/ALIAS/CREATE`, and `/KUIP/ALIAS/DELETE` are created.

Example for the creation of a help item

```
>Menu /KUIP
>Help_item FUNCTIONS
>Guidance
      ...
```

This example creates the help item `/KUIP/FUNCTIONS`.

Example for a guidance text

```
>Name MYDEF

>Menu MYMENU
>Guidance
This is the menu for my own commands

* The next directive signals the end of the guidance text.
>Help_item MYCMD
>Guidance
* Lines starting with * are ignored.
This is the first sentence of the first paragraph.  The HELP menus
show the first phrase of the first paragraph.  The phrase
finishes at the first line ending with a '.' which is only now.
The first paragraph is still continuing here but this
sentence does not appear in the HELP menu.
.
Consecutive lines are formatted into paragraphs.
The maximum line width used for terminal output is determined
by the SET/COLUMNS command (usually 80 columns).

Because blank lines are ignored the start of a new paragraph has to be
indicated by a line containing only a '.' in the first column.
.
A '.' at the end of a line indicates a sentence ending period and
adds an additional space for improved readability.
Abbreviations like e.g. or etc. should therefore not appear
at the end of a line.
For consistency the extra space between sentences belonging to
the menu phrase should be added by hand.

 Text not starting in the first column is verbatim
             material and will not be reformatted.
.
       This should       be used

                    for
          examples  and  tables.
Again extra space between verbatim lines has to be indicated by '.' lines.
For terminal output a verbatim block is always separated by
a blank line from the surrounding paragraphs.

>User_help MYUHLP
```

Resulting HELP menu

```
  /MYMENU

  This is the menu for my own commands


From  /MYMENU/...

 1:   MYCMD          This is the first sentence of the first paragraph.  The HELP menus
                     show the first phrase of the first paragraph.  The phrase finishes
                     at the first line ending with a '.' which is only now.
```

Formatted output for "MANUAL MYMENU -LATEX"

MYCMD

This is the first sentence of the first paragraph. The HELP menus show the first phrase of the first paragraph. The phrase finishes at the first line ending with a '.' which is only now. The first paragraph is still continuing here but this sentence does not appear in the HELP menu.
Consecutive lines are formatted into paragraphs. The maximum line width used for terminal output is determined by the SET/COLUMNS command (usually 80 columns). Because blank lines are ignored the start of a new paragraph has to be indicated by a line containing only a '.' in the first column.
A '.' at the end of a line indicates a sentence ending period and adds an additional space for improved readability. Abbreviations like e.g. or etc. should therefore not appear at the end of a line. For consistency the extra space between sentences belonging to the menu phrase should be added by hand.

```
    Text not starting in the first column is verbatim
                material and will not be reformatted.

          This should       be used
                       for
             examples  and  tables.
```

Again extra space between verbatim lines has to be indicated by '.' lines. For terminal output a verbatim block is always separated by a blank line from the surrounding paragraphs.

```
 This is the user help information for 'MYCMD'.

 Current date and time: 941004 1656
```

Example for a user help routine in Fortran

```
      SUBROUTINE MYUHLP
      CHARACTER CMD*32
*
      CALL KUHELP(LUN,CMD)
*
      WRITE(LUN,'(3A)')
     +  ' This is the user help information for '''
     +  ,CMD(:LENOCC(CMD)),'''.'
*
      CALL DATIME(IDATE,ITIME)
      WRITE(LUN,'(/A,I7,I5)')
     +  ' Current date and time:',IDATE,ITIME
      END
```

Terminal output for “HELP MYMENU”

```
  This is the first sentence of the first paragraph.  The HELP menus show the
  first phrase of the first paragraph.  The phrase finishes at the first line
  ending with a '.' which is only now.  The first paragraph is still
  continuing here but this sentence does not appear in the HELP menu.

  Consecutive lines are formatted into paragraphs.  The maximum line width
  used for terminal output is determined by the SET/COLUMNS command (usually
  80 columns).  Because blank lines are ignored the start of a new paragraph
  has to be indicated by a line containing only a '.' in the first column.

  A '.' at the end of a line indicates a sentence ending period and adds an
  additional space for improved readability.  Abbreviations like e.g. or etc.
  should therefore not appear at the end of a line.  For consistency the
  extra space between sentences belonging to the menu phrase should be added
  by hand.

   Text not starting in the first column is verbatim
               material and will not be reformatted.

         This should       be used
                     for
            examples  and  tables.

  Again extra space between verbatim lines has to be indicated by '.' lines.
  For terminal output a verbatim block is always separated by a blank line
  from the surrounding paragraphs.

This is the user help information for 'MYCMD'.

Current date and time: 941004 1656
```

```
>GUIDANCE
   text
   ...
```

The `>Guidance` directive defines the help text attached to the most recent command tree element created by a `>Menu`, `>Command`, or `>Help_item` directive. The help text ends at the next CDF directive. The formatting rules applied for terminal output and by the `MANUAL` command in LaTeX mode are explained in the example below.

```
>USER_HELP  user-help-routine
```

The `>User_help` directive may follow a `>Command` or `>Help_item` directive. The *user-help-routine* is called by the `HELP` command and allows to add runtime dependent information at the end of the static `>Guidance` text.

A Fortran *user-help-routine* has to call `KUHELP` in order to obtain the logical unit where the additional help information should be written to. The second argument of `KUHELP` returns the command or help item name for which `HELP` was requested.

A *user-help-routine* in C receives the command name as argument and has to return the guidance text as a NULL-terminated, allocated array of pointers.

Example for a user help routine in C

```
#include <stdlib.h>
#include <time.h>
#include "kstring.h" /* see chapter 4 for str2dup() etc. */

char** myUserHelp( const char* cmd_name )
{
  char** text = (char**)malloc( 4 * sizeof(char*) );

  struct tm *newtime;
  time_t ltime;
  time( &ltime );
  newtime = localtime( &ltime );

  text[0] = str3dup( " This is the user help information for '",
                     cmd_name, "'" );
  text[1] = strdup( "" );
  text[2] = str2dup( " Current date and time: ", asctime( newtime ) );
  /* asctime ends in a newline which we don't want */
  text[2][strlen(text[2])-1] = '\0';

  text[3] = NULL;
  return text;
}
```

```
>PARAMETERS
  mandatory-parameters
+
  optional-parameters
++
  constant-parameters
```

The `>Parameters` directive heads the parameter definition for a command. The lines following up to the next directive must be parameter definitions of the form

parameter-name *prompt-string* *parameter-type* [*attribute-list*]

The underscore continuation has to be used if the complete definition does not fit onto a single single. Mandatory and optional parameters are separated by a control line containing only a "+" character in the first column. If there are only mandatory parameters the "+" line is not required.

Non-positional arguments can be specified in a command line as *parameter-name*=*value*. To allow abbreviations for *parameter-name* the minimum length has to be indicated by a "*" character, e.g.

```
DIR*ECTORY    'Directory path'    C
```

defines a parameter `DIRECTORY` which can be assigned in a command line as "`DIR=`*value*".

The *prompt-string* is used to prompt for missing mandatory arguments and as labels in the Motif command panel. Possible *parameter-type* values are "`C`" for character strings, "`I`" for integer numbers, and "`R`" for real numbers.

The *attribute-list* for mandatory parameters can be empty. For optional parameters it must specify at least the default attribute:

```
D=default-value
```

For mandatory parameters the default value is only used when prompting for missing arguments and the reply is an empty line. The default value for optional parameters is passed to the action routine whenever the argument is missing or given as "`!`" on the command line.

For all parameter types `C`, `I`, and `R` the range attribute defines a comma-separated list of valid values:

```
R=value-1,value-2,...value-n
```

e.g.

```
PRIME    'Prime number below 10'    I    R=2,3,5,7
```

For option parameters there is an alternative way to specify the list of valid values together with an explanation text for each option (see below). For the numeric parameter types `I` and `R` the range attribute can have the form

```
R=lower-limit:upper-limit
```

to limit valid values to the interval including the boundaries. Either the *lower-limit* or the *upper-limit* may be omitted to indicate an unlimited interval in one direction.

Numeric parameters with a limited range are displayed with a slider in the Motif command panel. The slider step size for type "`R`" parameters is determined by the number of digits following the decimal point in the range definition, e.g.

```
PROB*ABILITY  'Detection probability'  R  D=0.5  R=0:1.00
```

Example for option parameter definitions

```
>Name MYDEF
>Menu MYMENU

>Command ECHO
>Action XECHO
>Parameters
STRING    'Mandatory string' C
+
NUMBER    'Small negative number'  I D=0    R=-10:0
MINUS     'Optional string'        C D=MN   Minus
POSITIVE  'Positive real'          R D=1.23 R=0:  Slider=1:2.00

OPTION    'Implicit option'        C D=' '
-          Blank option value
--         Option value '-' can make trouble.
-1         Same is true for digits.

CHARSET   'Self documenting option values in Range attribute' _
                                C Option  D=NATIVE  R=NATIVE,ASCII,EBCDIC

FORMAT    'Formatting rules'    C Option  D=LINE
-LINE      The full explanation text has to be on a single logical line. _
           The underscore can be used to join several physical lines _
           into one logical line.
-BLANKS    Leading blanks are ignored. _
           Indentation can be used to make the CDF more readable. _
           Multiple blanks are replaced by a single blank.

>Guidance
Echo command line arguments to demonstrate exception rules
for option assignments.
```

creates a slider for the values 0.00, 0.01, ..., 1.00. If the range is unlimited or too large the `Slider` attribute

`SLIDER=`*`lower-limit`*`:`*`upper-limit`*

allows to define a restricted range which affects the slider only. E.g.

```
EFF*ICIENCY  'Reconstruction efficiency'  R  R=0:1  Slider=0.900:1
```

creates a slider for the values 0.900, 0.901, ..., 1.000.

The `Option` attributes defines a parameter to be an option for which "`-`*value*" can be used to specify non-positional option arguments on the command line. The `Option` attribute is set implicitly if *parameter-name* begins with `OPTION` or `CHOPT`. The "`-`" mechanism requires a range definition including all valid option values. The list of option values is defined by lines

`-option-value     explanation-text`

following the option parameter definition itself. The "`-`" character must be in the first column and is not part of the option value. The underscore continuation has to be used if the complete *explanation-text* does

Argument assignments for command ECHO

```
1.  2: echo  Number:-1  -1
NUMBER= -1  MINUS=MN     POSITIVE=1.23  OPTION=       CHARSET=NATIVE   FORMAT=LINE

1.  3: echo  -1  Option:1
NUMBER=  0  MINUS=MN     POSITIVE=1.23  OPTION=1      CHARSET=NATIVE   FORMAT=LINE

1.  4: echo  Option:-1  !  !  !  -1
NUMBER=  0  MINUS=MN     POSITIVE=1.23  OPTION=-1     CHARSET=NATIVE   FORMAT=LINE

1.  5: echo  -line  Format:LINE
NUMBER=  0  MINUS=MN     POSITIVE=1.23  OPTION=       CHARSET=NATIVE   FORMAT=LINE

1.  6: echo  Minus:-LINE  !  -LINE
NUMBER=  0  MINUS=-LINE  POSITIVE=1.23  OPTION=       CHARSET=NATIVE   FORMAT=LINE

1.  7: echo  -line  -ascii  Charset:ASCII/Format:LINE
NUMBER=  0  MINUS=MN     POSITIVE=1.23  OPTION=       CHARSET=ASCII    FORMAT=LINE
```

not fit onto a single single. The rules applied for formatting the list of options as a table are explained in the example.

A command can contain contain several option parameters as long as the option values are distinct. Option values can be specified with the range attribute if an *explanation-text* is not required. Blank is permitted as *option-value* but cannot be combined with any other value. Consequently it should only be used if all option values are mutually exclusive, and it should be the default value.

The interpretation of a "`-`*value*" argument as option is considered only if *value* consists exclusively of valid values from a single option parameter. In some instances "`-`*value*" is intended be a positional argument even though it could match an option parameter as well. It depends then on the definition of the parameter which should be assigned next whether the argument is actually interpreted as option "*value*".

If "`-`*value*" can be interpreted as a negative number and the next parameter is numeric the option assignment is inhibited. A character parameter can be given the `Minus` attribute to indicate that arguments at its position should never be assign to an option parameter.

The leading "`-`" is usually stripped off in an option assignment. Only in case the next argument position belongs to the option parameter itself and "`-`" is one of its option values it is passed on to the action routine. In order to avoid confusion due to this position dependence digits and the minus sign should not be used as option values.

It is often desirable for a command to be able to accept a variable-length list of values for one of the parameters. If the command operation is independent for each list item the `Loop` attribute can be added to the corresponding parameter definition. For a comma-separated argument list the action routine is then called repeatedly passing it the next list item each time. E.g. with the definition

```
>Command DELETE
>Parameters
LIST  'List of items'  C    Loop
```

the command line

```
DELETE x,y,z
```

is equivalent to

```
DELETE x ;  DELETE y ;  DELETE z
```

and the action routine only needs to handle a single item. A "`,`" inside a quoted string or an unbalanced set of parentheses is not treated as item separator, i.e. "`'foo,bar'`" and "`xyz(3,4,5)`" are single items only.

In case the action routine needs access to all list items it can call `KUGETL` to retrieve the items one by one. The `Vararg` attribute can be given to a list parameter if it is the last one of the command. This allows to use `KUGETL` also for a blank separated item list, i.e. with the definition

```
>Command ASSIGN
>Parameters
ARRAY  'Array name'  C
VALUES  'List of values'  C  Vararg
```

both command line forms

```
ASSIGN  name  foo,bar
ASSIGN  name  foo  bar
```

can be handled by the same action routine.

In Motif mode the arguments entered in the command panel are usually treated as a single value, i.e. if they contain a blanks they are implicitly quoted. In some cases, for example the macro arguments passed in the `EXEC` command, this behavior must be inhibited. The `Separate` attribute allows to flag the last parameter that the value entered in the Motif input field should be interpreted as separate command line tokens.

```
>ACTION  action-routine
```

The `>Action` directive must follow a `>Command` and defines the routine which should be called after decoding the command line in order to perform the intended operations.

An action routine does not have arguments. The command line arguments have to be retrieved by calling the suitable `KUGET`*x* routine for each parameter from inside the action routine.

The same action routine can be used for different commands. The routines `KUPATH` and `KUPATL` allow to identify which command was actually requested.

Example for an action routine

```
      SUBROUTINE XECHO
      CHARACTER*80 STRING
      CHARACTER*6 CMINUS, OPTION
      CHARACTER*8 CHRSET, FORMAT

      CALL KUGETS(STRING, LSTR)
      CALL KUGETI(NUMBER)
      CALL KUGETC(CMINUS, LMIN)
      CALL KUGETR(POSTIV)
      CALL KUGETC(OPTION, LOPT)
      CALL KUGETC(CHRSET, LCHS)
      CALL KUGETC(FORMAT, LFMT)

      PRINT '(A,I3,3A,F4.2,6A/)',
     +  ' NUMBER=',NUMBER,
     +  '  MINUS=',CMINUS,
     +  ' POSITIVE=',POSTIV,
     +  '  OPTION=',OPTION,
     +  ' CHARSET=',CHRSET,
     +  ' FORMAT=',FORMAT

      END
```

3.3 Browser Concepts and Definitions

The high-light of the KUIP/Motif interface is a general-purpose object browser. Before going into the details of the CDF directives for customizing the browser we want to introduce the main concepts and the terminology used.

The browser allows to traverse a hierarchical directory structure of objects. Operations on these objects can be chosen from pop-up menus which depend on the object type (or "class").

To incorporate your own application specific objects into the browser you have to:

– describe in the CDF ("Command Definition File") the object types ("classes") and the containers for these objects ("browsables"),
– write "small" routines to scan through the list of objects, and eventually the list of browsables,
– describe in the CDF the action menus for classes of objects and browsables.

3.3.1 Classes of Objects

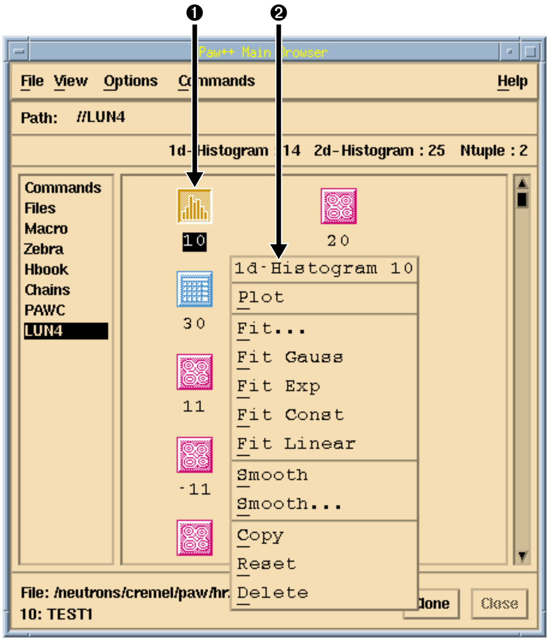
"Classes" of objects can be any kind of entities handled by an application. For example:

– HIGZ pictures and HBOOK data (1d-histograms, 2d-histograms, N-tuples, chains and directories) are "Classes" defined in PAW++.
– in Geant++ the classes are the data structures: volumes, materials, particles, ...
– in KUIP/Motif the commands themselves have been defined as classes, as well as the different type of files (read, read-write, executable, and macros ...).

Object classes are defined in the CDF with the directive:

```
>Class  class-name  menu-title  [ big-icon  small-icon ]
```

E.g. The directive “`>Class 1d '1d-Histogram' big_1d sm_1d`” in the PAW CDF is used for 1D histograms : the class name is “1d”, “big_1d” (normal size) and “sm_1d” (small size) are the icons used by the browser for a graphical representation and “1d-Histogram” is the title text for the pop-up menu displayed when object types “1d” are selected in the browser (see figure 3.2).



CDF description (extract) :

Example of the “Class” directive

```
>Class 1d 1d-Histogram big_1d sm_1d
 Plot          .  .  default_action%C
'/Fit...'      .  'Histo/Fit [this]'
'Fit Gauss'    .  'Histo/Fit [this] G'
'Fit Exp'      .  'Histo/Fit [this] E'
'Fit Const'    .  'Histo/Fit [this] P0'
'Fit Linear'   .  'Histo/Fit [this] P1'
 /Smooth       .  'Smooth [this]'
'Smooth...'    .  '-Smooth [this]'
 /Copy         .  'Histo/Copy [this]'
 Reset         .  'Histo/Op/Reset [this]'
!Delete        .  'Hio/Hscratch [this]'
+
...
```

❶ Icon used to represent the object class “1d” (normal size).

❷ Action menu (title “1d-Histogram”) associated to the selected object (histogram of class “1d” with name/identifier “10”).

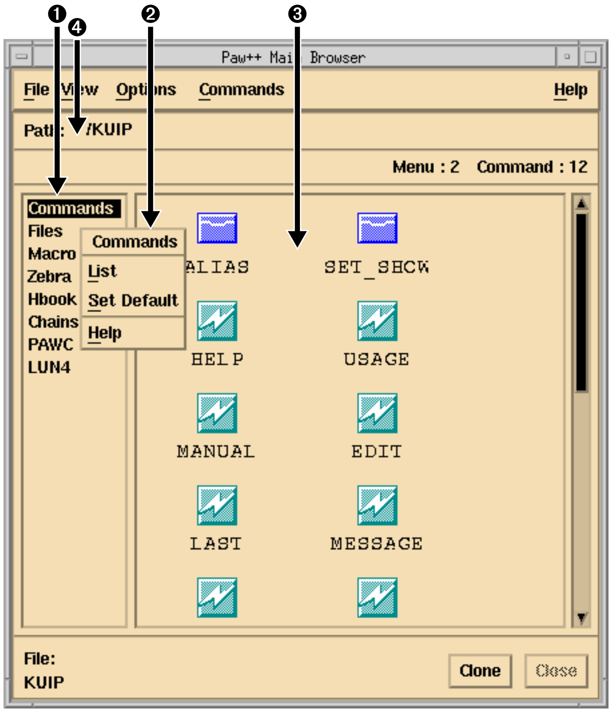
Figure 3.2: Object Classes

3.3.2 Browsables

All objects are part of container objects constituting the top level directory (e.g. ZEBRA/RZ files.). We will call these containers the “browsables”. They are defined with the CDF directive

```
>Browse  browsable-class  menu-title  scan-objects  [ scan-browsables ]
```

For example: in PAW++, a browsable HBOOK (see figure 3.3) has been defined, which is a container for any kind of HBOOK data (1d/2d histograms and Ntuples). In KUIP/Motif, the browsable “Files” contains and displays all files (read, read-write, executables, and KUIP macros). Another browsable “Commands” has been defined for all the commands and menus defined by KUIP and the application. For a detailed description of the “`>Class`” and “`>Browse`” CDF directives see sections 3.4.1 and 3.4.2.



CDF description (extract) :

Example of the "Browse" directive

```
>Browse Commands . kscncmds%C
 List
'Set Default'  .  ' Set/Root /'
...
>Class /Menu Menu big_menu sm_menu
...
>Class Cmd Command big_cmd sm_cmd
...
```

❶ The browsable "Commands" is selected.
❷ Action menu corresponding to this browsable.
❸ **"Object window"** with the list of commands (class "Cmd") and menus (class "/Menu") for the current directory "/KUIP" ❹.

Figure 3.3: Browsables

3.3.3 The "scan-objects" routine

To make the objects belonging to one browsable for a given path (directory) accessible by the browser, the application has to provide a scanning routine which, when called for the first time, returns the first object and subsequently returns the next object until no more objects are left: this is what we have called the "scan-objects" routine in the CDF directive for the browsable definition. The browser passes to this routine the identification of the requested subdirectory and expects in return:

- the object name, e.g.:
  - for "`>Browse HBOOK`" in PAW++: the histogram or Ntuple identifier (number).
  - for "`>Browse Commands`" in KUIP/Motif: the menu or command name.
- the object class, e.g.:
  - for "`>Browse HBOOK`" in PAW++: "1d", "2d" or "Ntuple" (assuming that "`>Class 1d`", "`>Class 2d`" and "`>Class Ntuple`" have been defined).
  - for "`>Browse Commands`" in KUIP/Motif: "Menu", "Cmd" and "InvCmd" (assuming that "`>Class Menu`", "`>Class Cmd`" and "`>Class InvCmd`" have been defined).
- and eventually a short and a long description text (title) of this object (which will be used to display the list of objects in various forms: big icons, small icons, text only).

This "scan-objects" routine can be coded in Fortran or in C.

3.3.4 The “scan-browsables” routine

We have to make a distinction between two kinds of browsables:

- “single instance” browsables,
- or “multiple instances” browsables.

A “multiple instances” browsable gives the possibility to have several instances of the same browsable. For instance, in PAW++, the browsable “`>Browse HBOOK`” which is defined for the ZEBRA/RZ files, acts as a container for the objects 1d/2d histograms and Ntuples. As it is possible to have several ZEBRA/RZ files opened at the same time and connected to different logical units (LUN), the browsable HBOOK has been defined as a “multiple instances” one. This is done by providing a second scanning routine from which the name of all the currently connected instances of a given browsable can be retrieved (e.g. for all the ZEBRA/RZ files opened and connected to some logical unit the name is a concatenation of “LUN” and this number, i.e. “LUN8” for logical unit 8). This is what we have called the “scan-browsables” routine (optional for a “single instance” browsable) in the CDF directive for the browsable definition.

This “scan-browsables” routine can be coded in Fortran or in C.

Note that “single instance” browsables are always displayed in the browser (as soon as it is created) whereas “multiple instances”browsables can be accessed and displayed at run time (e.g. in PAW++ when a new ZEBRA/RZ file is opened).

For a more precise description of the calling sequence of the “scan-objects” and “scan-browsables” routines see section 3.4.2.

3.3.5 Directories

A tree structure of objects can easily be achieved by defining a special class for subdirectories of a browsable. The corresponding CDF directive is the same as for simple object classes, “`>Class`”, except that there must a “`/`” in front of the class name.

For example:

- in PAW++, ZEBRA/RZ directories (browsable “HBOOK”) are “directories” classes of objects:

```
>Class /dir Directory big_dir sm_dir
```

  Ntuple chains (in the “Chains” browsable) are also defined as a “directory” class of objects:

```
>Class /chain Chain big_chain sm_chain
```

- in KUIP/Motif the menus (in the “Commands” browsable) are defined as a “directory” class of objects:

```
>Class /Menu Menu big_menu sm_menu
```

  This is also true for directories of files (in the “File” browsable):

```
>Class /DirFile Directory big_menu sm_menu
```

This "special" class of objects obeys to certain rules:

- The first item in the action menu is always "List" and means "switch into this subdirectory". Selecting this item (automatically performed with a <double click> with the left button) has the effect to update the content of the **"Object window"** with the list of objects contained into the new subdirectory.
- The entry "Path:" (Fig. 3.3, ❹) is automatically updated with the new directory path, which is formed by concatenating the previous path, a "/", and the name of the directory object selected. Going upwards in the directory hierarchy is done by selecting a substring of the current path displayed in this text entry. Clicking a second time on the same path segment performs the directory change and updates the **"Object window"** in the same way as selecting the "List" menu entry for a directory object.

3.3.6 Action menus

For each class of objects it is possible to define two menus which describe possible actions if an object of this class is selected:

- (1)either in the browser **"Object window"**,
- (2)or identified inside a HIGZ **Graphics Window** (if any).

For each browsable it is possible to define:

- (3)one menu which describes possible actions if this browsable is selected (**"Browsable window"**),
- (4)for "multiple instances" browsables, a second menu which describes the actions required to connect or de-connect one instance of this browsable at run time (menu "File" in the **"Main Browser"**).

Menus (1), (2) and (3) pop up when pressing the right mouse button in the corresponding window, and the selected action is performed when the button is released. A <double click> with the left mouse button on one specific object or on one specific browsable always executes the first menu item. Menu (4) is accessed by selecting the entry "File" in the **"Main Browser"** menu-bar.

Object and browsable specific action menus are derived from sequences of action definitions written in the CDF and following the `>Class` or `>Browse` directive (Fig. 3.2 and 3.3). For a precise description of the syntax and behavior of the CDF directives for action menus see section 3.4.3.

3.4 CDF directives for KUIP/Motif

We will describe in this section the new CDF (Command Definition File) directives which have been introduced in KUIP/Motif. Please note that these new directives have to be used only to take advantage of the MOTIF style and to include your application specific objects into the interface (and especially into the browser(s)). BUT one must be aware that IT IS POSSIBLE to get a Motif interface for any KUIP based application WITHOUT modifying the CDF, but only modifying very slightly the application main program in order to give control to the "Motif main loop" (i.e. call KUWHAM instead of KUWHAT or KUWHAG).

In all the directives we will describe the square brackets ("[]") mean that the corresponding entity is optional: either some default value is automatically provided if it is missing (e.g. for titles), or the entity is not necessary requested. If such an optional entity happens to be in the middle of the directive (not at the end) then it has to be replaced by a "." if no real value is given.

We advice users who want to take advantage of the Motif style to compile their CDFs into C code (instead of Fortran): this is automatically performed by the KUIPC compiler when giving the extension "`.c`" to the output file which is produced.

N.B. It is preferable to separate the directives for the usual command mode interface and the directives specific to the Motif style (objects, browsables and action menus definitions) into different CDFs. Then it will be easy and possible to make the same application work on a dumb terminal (offering only the KUIP command mode interface) or on a Motif workstation, with only very few differences in the application main program.

3.4.1 For Classes of Objects

As we have already seen (section 3.3.1), classes of objects are defined in the CDF with the directive

```
>Class  class-name  menu-title  [ big-icon  small-icon ]
```

for example,

```
>Class ExFile 'Executable File' big_fx sm_fx
```

The "menu-title" should be a short explanation concerning the class (or type) of object and is used as a title text for the action menu which is displayed when one object of this class is selected (either in the browser, or in the **Graphics Window** if any). If no title is given the "class-name" is used instead.

"big-icon" and "small-icon" are the names for the icons representing graphically this class of object inside the browser. Default icons are provided if these values are missing, but then the same iconic representation will be used for all object classes. It is a lot better to have a different representation for each object class. To create a new icon bitmap you can use the X11 standard bitmap editor. The icons definitions in the CDF follow the directive `>Icon_bitmaps` (see 3.4.5.3).

3.4.2 For Browsables

As we have seen in section 3.3.2, browsables are defined with the CDF directive

```
>Browse  browsable-class  menu-title  scan-objects  [ scan-browsables ]
```

for example,

```
>Browse Macro . kmbmac%C kmbmdi%C
```

The "menu-title" should be a short explanation on the browsable itself and is used as a title text for the action menu which is displayed in the browser "FileList" window when this browsable is selected. If a "." is put instead, then the browsable-class becomes by default the menu-title.

3.4.2.1 The “scan-objects” Routine

The “scan-objects” routine can be written in Fortran or in C. The browser passes to this routine the identification of the requested subdirectory (path) and expects in return: the object name, the class name, and eventually a short and a long description text (title) of this object. In both cases (Fortran or C) the calling sequence is predefined.

In Fortran the calling sequence of the “scan-objects” routine is the following:

scan-objects routine (in Fortran)

```
      SUBROUTINE SCNOBJ(BRNAME,BRCLAS,BRPATH,OBNAME,OBCLAS,STEXT,LTEXT)
      CHARACTER*(*) BRNAME,BRCLAS,BRPATH,OBNAME,OBCLAS,STEXT,LTEXT
*
*            Browser interface to return next object
*
*            Input Parameters  :
*
*                      BRNAME  : browsable name (displayed in the browser)
*                      BRCLAS  : browsable class name (from the ‘‘>Browse’’ directive)
*                      BRPATH  : current directory path
*
*            Input / Output    :
*
*                      OBNAME  : object name or identifier
*                                ’ ’ the first time, previous object otherwise
*
*            Output Parameters :
*
*                      OBCLASS : class name
*                      STEXT   : short text description of the object
*                      LTEXT   : long text description of the object
*
*     ...
*     IF(OBNAME.EQ.’ ’) THEN
*        Initialize the scan-objects routine
*     ENDIF
*     ...
*
      OBCLASS = ...
      STEXT = ...
      LTEXT = ...
*
      END
```

In C the calling sequence is:

scan-objects routine (in C)

```
char **scnobj( brobj_name, brcls_name, bpath, n )
     char *brobj_name;  /* browsable name (displayed in the browser) */
     char *brcls_name;  /* browsable class (from the ‘‘>Browse’’ directive) */
     char *bpath;       /* current directory path */
     int n;             /* object position (0 the first time) */
{
  static char     *obj_desc[4];
```

```
    ...
    if (n == 0) {
       /* Initialize the scan-objects routine */
    }
    ...

    return obj_desc;   /* obj_desc[0] --> object name or identifier
                          obj_desc[1] --> class name
                          obj_desc[2] --> short text description of the object
                          obj_desc[3] --> long text description of the object
                        */
}
```

The browsable name and class-name are identical for "single instance" browsables. They can be different for "multiple instances" browsables. e.g. for "`>Browse HBOOK`" in PAW++: the browsable class is "HBOOK" and the browsable name is "LUN8" for a ZEBRA/RZ file connected to logical unit 8.

The selected object name (or identifier) and its corresponding long text description are printed at the very bottom line of the browser (❻ in Fig.2.11).

3.4.2.2 The "scan-browsables" Routine

The "scan-browsables" routine (optional for "single instance" browsables but required for "multiple instances" ones) can be written in Fortran or in C. The browser passes to this routine the browsable class-name (e.g. in PAW++, "`HBOOK`" for ZEBRA/RZ files) and expects in return: the browsable name (e.g. "`LUN8`" for a ZEBRA/RZ file connected to logical unit 8) and eventually a string with the description of a predefined set of variables concerning this browsable. These variables are used as "key-words" in the actions menus connected to the browsables or the classes of objects. Some of these variables are fixed by KUIP/Motif itself:

– the variable "file" is used to fill the corresponding label ("File:") at the bottom of the browser.
– "root" is used for setting the path of the top level directory (e.g. "`//LUN8`"),

The 'scan-browsables" routine for a "single instance" browsable can be defined just for setting this "description string" with the required key-words.

In Fortran the calling sequence of the "scan-browsables" routine is the following:

scan-browsables routine (in Fortran)

```
      SUBROUTINE SCNBRO(BRCLAS,BRNAME,VARSET)
      CHARACTER*(*) BRCLAS,BRNAME,VARSET
*
*          Browser interface to return next browsable instance
*
*          Input Parameters  :
*
*                 BRCLAS  : browsable class name (from the ``>Browse'' directive)
*
*          Input / Output    :
*
```

```
*                     BRNAME  : browsable name (displayed in the browser)
*                               ' ' the first time.
*
*            Output Parameters :
*
*                     VARSET  : set of variables for this browsable.
*
*     First call to this routine
      IF(BRNAME.EQ.' ') ...
*
      ...

      BRNAME = ...
      VARSET='root= ... file= ... ...=...'
      ...
*
      END
```

E.g. in PAW++, for ">Browse HBOOK" and the browsable "LUN8" (ZEBRA/RZ file connected on logical unit 8) the routine SCNBRO will return something like:

```
VARSET = root=//LUN8 file='/user/paw/demo/hrztest.dat'  unit=8
```

In this example "unit" is a key-word specific to the browsable HBOOK that can be used later on in the action menu definition (e.g. "Close [unit]"). (See next section 3.4.3 for a complete description of the "action menu definition").

In C the calling sequence is the following:

scan-browsables routine (in C)

```
char **scnbro( class_name, first )
     char *class_name;
     int first;
{
    static char *path_desc[2];

    ...
    if (first)
       /* First call to this routine */

    ...

    return path_desc;  /* path_desc[0] --> browsable name
                          path_desc[1] --> set of variables for this browsable
                        */
}
```

3.4.3 For Actions Menus

Both the "`>Class`" and "`>Browse`" directives can be followed by sequences of action definitions which obey to the following syntax:

```
menu-text  [ special-flags ]  [ action-string ]  [ action-routine ]
```

N.B. the components which are missing, when optional (enclosed with "[]" in the syntax description above), must be replaced by a "." if they are not the last ones.

The "menu-text" is the text that will appear in the pop-up menu. It should be short but meaningful. "/" and "!", when they are the very first character of this menu-text have special meanings:

- a "/" means that the menu item has to be preceded by a separator,
- a "!" means that the browser has to be updated when the action is performed (because the "objects window" may have changed).

At the moment, the entity "special-flags" is only used for a "Toggle" representation of the menu-item, instead of a normal "push-Button" representation. This gives the possibility to define special "menu text" and give them a "toggle" behavior (see section 3.4.3.3).

The action can be specified as a string of commands ("`action-string`") which should be executed and/or a user-written and application specific routine ("`action-routine`") which should be called.

In both cases ("`>Class`" and "`>Browse`") two sets of menus can coexist and are separated by a blank line starting with the character "+". The first menu always applies to objects/browsables which are identified in the browser. In most cases the menu pops up when pressing the right mouse button and the selected action is performed when the button is released.

For the "`>Class`" directive the second menu applies to objects which are identified in the HIGZ **Graphics Window** if any. (E.g. in PAW++, "`>Class 1d`" and "`>Class 2d`" histograms can be identified either in the browser or in the graphics window: two sets of menu are then defined in the CDF).

For the "`>Browse`" directive a second menu can be defined to fill the "File" menu-bar entry in the browser with the list of commands to connect or de-connect one instance of a "multiple instances" browsable (e.g. in PAW++ open or close a ZEBRA/RZ file containing HBOOK data).

Only one of the two menu sets can be defined in both cases.

N.B. For graphical object identification inside the graphics window please refer to the description of the two HIGZ routines IGPID and IGOBJ in the HIGZ manual [9].

3.4.3.1 Action-string (String of Commands)

When the action is specified as a string of commands, constructs of the form "`[`*var*`]`" can insert identifications of the selected object in the command string:

- `[this]` is replaced by the object name, e.g. in PAW++: histogram "10".
- `[that]` is replaced by the short description text returned by the scanning routine and can thus be used as an alias name.
- `[name]` is replaced by the name of the browsable, e.g. in PAW++, "`LUN8`" for an ZEBRA/RZ file opened on unit 8.

– [root] is replaced by the path of the top level directory, e.g. in PAW++, "//LUN8".
– [path] is replaced by the complete path to the directory in which the object is contained, e.g. in PAW++, "//LUN8/MYDIR". Initially, [path] is set to [root].

As we have seen in the previous section (3.4.2.2) the replacement values for [name], [root] and, eventually, additional variables specific to the application (e.g. [unit] in PAW++ for the browsable HBOOK) have to be returned by the application specific "scan-browsables" routine. For single instance browsables, when this routine (optional) is not defined, [name] is substituted by the class name and the field *menu-title* of the >Browse directive is used as the definition of [root].

3.4.3.2 Action-routine

If the action corresponding to some menu-text is specified as an action-routine, it can be coded either in Fortran or in C. Fortran is the default, for C coding the two characters "%C" must be appended at the end of the routine name.

The predefined calling sequence of the action routine in Fortran depends on which entity the action menu applies:

– Case 1: menu defined for browsables (in the **"Browsable window"**).

Action-routine Skeleton (Fortran)

```
      SUBROUTINE ACTION(BRNAME,BRCLAS,BRPATH)
      CHARACTER*(*) BRNAME,BRCLAS,BRPATH
*
*     Write application code ...
*
      END
```

– Case 2: entry in the menu "File" (to connect a new browsable instance).

Action-routine Skeleton (Fortran)

```
      SUBROUTINE ACTION(BRCLAS)
      CHARACTER*(*) BRCLAS
*
*     Write application code ...
*
      END
```

– Case 3: menu defined for objects which are identified in the browser **"Object window"**.

Action-routine Skeleton (Fortran)

```
      SUBROUTINE ACTION(BRNAME,BRCLAS,BRPATH,OBNAME,OBCLAS,STEXT,LTEXT)
      CHARACTER*(*) BRNAME,BRCLAS,BRPATH,OBNAME,OBCLAS,STEXT,LTEXT
*
*     Write application code ...
*
      END
```

– Case 4: menu defined for the objects identified in the graphics window.

Action-routine Skeleton (Fortran)

```
      SUBROUTINE ACTION(OBNAME,OBCLAS)
      CHARACTER*(*) OBNAME,OBCLAS
*
*     Write application code ...
*
      END
```

In all cases the meaning of the input parameters is exactly the same as for the "scan-objects" routine (section 3.4.2.1):

```
BRNAME  : browsable name (displayed in the browser)
BRCLAS  : browsable class name (from the ``>Browse'' directive)
BRPATH  : current directory path
OBNAME  : object name or identifier
OBCLASS : class name
STEXT   : short text description of the object
LTEXT   : long text description of the object
```

The C calling sequence is the same as for Motif call-back routines:

Action-routine Skeleton (C)

```
#include <stdio.h>
#include <Xm/Xm.h>

typedef enum {
  BRACT_OPEN = 0,  BRACT_ROOT = 1,
  BRACT_CONT = 2,  BRACT_GRAF = 3
} BrActTag;

typedef struct _BrClientdata {
  BrActTag    tag;
  char       *brobj;
  char       *brcls;
  char       *path;
  char       *kmobj;
  char       *kmcls;
  char       *stext;
  char       *ltext;
  char       *mtext;
} BrClientdata;

void action-routine( Widget w,
                     BrClientdata *client_data,
                     XmAnyCallbackStruct *cbs )
{
   /* Write application code ... */
}
```

The input parameter `client_data` contains the relevant information for an entity inn a given action menu:

```
tag   ---> BRACT_OPEN (menu ``File'' in the browser) ,
           or BRACT_ROOT (``Browse'' menu),
           or BRACT_CONT (``Class'' menu in the browser),
           or BRACT_GRAF (``Class'' menu in the graphics window).
brobj ---> browsable name (displayed in the browser)
brcls ---> browsable class name (from the ``>Browse'' directive)
path  ---> current directory path
kmobj ---> object name or identifier
kmcls ---> class name
stext ---> short text description of the object
ltext ---> long text description of the object
mtext ---> menu text
```

3.4.3.3 Menu Items with "Toggle" Behavior ("special-flags")

In any kind of application, commands with only one parameter which can have two possible values (generally 1/0, On/Off or or TRUE/FALSE) are quite common. The usual way to represent such commands with Motif is to use "toggle buttons". It is possible, in the actions menus definition, to specify such menu items by using the "special flag" "T" or "T=0", "T=1" (entity "special-flags" in the syntax description made in section 3.4.3). In that case the same "menu-text" definition, with `[ special-flags ]` set to T (or T=0 or T=1), has to be written twice: the first time the `[ action-string ]` and/or `[ action-routine ]` must correspond to the transition state "On –> Off", while the second time they must correspond to the inverse, i.e. the transition state "Off –> On". Specifying "T=0" in the first line means that the initial state is "Off" (the toggle button is not pressed), whereas specifying "T=1" means that the initial state is "On" (the toggle button is pressed and visible).

3.4.3.4 Example

The following is an example of the action menus definition for the object class "1d" in PAW++ (1-dimensional histogram):

Example of action menus definition

```
>Class 1d 1d-Histogram big_1d sm_1d
 Plot                         .  . default_action%C
'/Fit...'                     .  'Histo/Fit [this]'
'Fit Gauss'                   .  'Histo/Fit [this] G'
'Fit Exp'                     .  'Histo/Fit [this] E'
...
'Smooth...'                   .  '-Smooth [this]'
 /Copy                        .  'Histo/Copy [this]'
 Reset                        .  'Histo/Op/Reset [this]'
!Delete                       .  'Histo/Hio/Hscratch [this]'
+
'Fit...'                      .  'Histo/Fit [this]'      default_G_action%C
'Fit Gauss'                   .  'Histo/Fit [this] G'    default_G_action%C
```

```
...
'Filled Lego'                 .  'Histo/Plot [this] LEGO1' default_G_action%C
'Default'                     .  'Histo/Plot [this]'       default_G_action%C
...
'Test on/off'               T=1 'Message off'
'Test on/off'               T   'Message on'
'Logarithmic Scale in X'    T=0 'OPT LINX'
'Logarithmic Scale in X'    T   'OPT LOGX'
```

As the object “1d” can be identified both in the browser (for the browsable HBOOK) and in the HIGZ **Graphics Window** two sets of menus are defined separated with a blank line starting with “+”.

By default the commands in the action string are executed immediately, provided that all mandatory arguments are present. If any mandatory argument is missing the corresponding “Command Argument Panel” comes up where they can be filled in. This default behavior can be modified in several ways:

- At run-time, pressing the `CTRL`-key when popping up the menu forces the command panel even if all mandatory arguments are specified.
- Putting a “`-`” in front of the command name (“`action-string`” definition in the CDF) forces the “Command Argument Panel” as well.
- Putting a “`+`” in front of the command name never produces a command panel, independently of the state of the `CTRL`-key.

The calling sequence of the action routines “default_action” and “default_G_action” (coded in ANSI C) is the following:

action-routine calling sequence (Example)

```
void default_action(Widget w, BrClientdata *client_data,
                    XmAnyCallbackStruct *cbs)
{
   /* Write application code ... */
}

void default_G_action(Widget w, BrClientdata *client_data,
                      XmAnyCallbackStruct *cbs)
{
   /* Write application code ... */
}
```

Both menu items “Test on/off” and “Logarithmic Scale in X” correspond to “toggle button behavior”. For “Test on/off” the initial state is “On” (toggle is pressed and visible) and the command to be executed to pass from “on” to “off’ is “Message off”. For “Logarithmic Scale in X” the initial state is “off” (linear scale) and the command to be executed to pass from “Logarithmic” (on) to “Linear” (off) is “OPT LINX”.

Picture 3.4 shows the output from KUIP/Motif when defining the action menu described above in the graphics window.

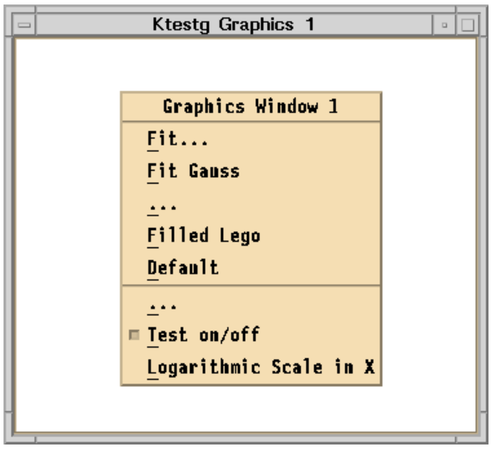


Figure 3.4: Example of action menu output from KUIP/Motif

3.4.4 For Graphical Objects Identification

A pop-up menu of actions can easily be defined in the CDF for the objects that are graphically "pick-able" in the **Graphics Window** (if any) with the HIGZ routine IGOBJ [9].

The only thing to do is to define a "Class" for this object, and define for this class the second action menu which applies to the graphics window. If this type of object does not appear in the browser it is not necessary to define for this class neither the first menu nor the icon representation.

E.g. in PAW++ such classes exist for the x- and y-axis:

X- and Y- axis Definition in PAW++ CDF

```
>Class x-axis 'X Axis'
+
'Logarithmic'                          . 'Option LOGX ; Histo/plot [this]'
'Linear'                               . 'Option LINX ; Histo/plot [this]'
'/Sort in alphabetical order'          . 'Sort [this] AY ; Histo/plot [this]'
...
'Character Font...'                    . '-Set VFON ; Histo/plot [this]'
'Axis Color...'                        . '-Set XCOL ; Histo/plot [this]'

>Class y-axis 'Y Axis'
+
'Logarithmic'                          . 'Option LOGY ; Histo/plot [this]'
'Linear'                               . 'Option LINY ; Histo/plot [this]'
'/Sort in alphabetical order'          . 'Sort [this] AY ; Histo/plot [this]'
...
'Character Font...'                    . '-Set VFON ; Histo/plot [this]'
'Axis Color...'                        . '-Set YCOL ; Histo/plot [this]'
```

3.4.5 For Customizing Your Application

Some CDF directives have been defined in order to facilitate and perform the customization of any application as well as possible. For instance the default size and positions of the windows opened by KUIP/Motif (“Executive”, “Browser” and eventually the “Graphics” window) may not be suitable for your own application. It is also a lot better to use your own iconic representation for the application defined objects rather than the default ones. You may also want to add your own buttons or menu items in the **“Executive Window”** (KXTERM).

On another hand, if the interface offered by KUIP/Motif is not sufficient for your application and if you want to get involved in Motif programming you are free to do so. This is what is done for example in PAW++ for the “Histogram Style Panel” and the “Ntuple Viewer” which are written directly with Motif.

All the directives which are described in this section (apart from the “`>Graphics`” one if you want a HIGZ graphics window) are not at all mandatory and have to be used only if you want to give a more “professional” look to your own application. It can also make your life easier if you really want to play with Motif !

3.4.5.1 Graphical Window Managed by HIGZ

KUIP/Motif can be interfaced to the X11 version of the HIGZ graphics package for an application which requires high level graphics. In order for the two packages to cooperate properly, applications using HIGZ have to add the CDF directive

```
>Graphics
```

at the beginning of the CDF and call `KUINIM` before calling `IGINIT`.

HIGZ allows one to store the name and the class for every displayed object by calling the routine `IGPID`. From this information KUIP/Motif can identify the object at the mouse pointer position and pop up the second set of action menus defined with the `>Class` directive (see section 3.4.5.1). This mechanism is extensively used in the Motif versions of GEANT (Geant++) and PAW (Paw++).

3.4.5.2 Buttons and Menu(s) Definition

An application can create its own buttons and pull-down menus either in the **“Executive Window”** (KXTERM) or the main browser window using the CDF directive

```
>Button menu-title button-label action-routine mnemonic
        accelerator accelerator-text [flag]
```

- *menu-title* is the title of the pull-down menu where the button has to be put. If it does not exist, a new pull-down menu with this title is created.
- *button-label* is the text label of the menu entry to be created.
- *action-routine* is the associated action automatically called for a “Button Press” event. It can be a string of commands or a routine coded in C or in Fortran 77. (If the routine is in C the name has to be ended with “`%C`” in the CDF declaration).
- *mnemonic* is a mnemonic definition for the label.

– *accelerator* and *accelerator-text* are accelerator definition for the label.
– if *flag* = "BR" menus/buttons are created in the main browser (instead of the executive window, by default).

3.4.5.3 Icon Bitmaps Definition

The CDF directive

```
>Icon_bitmap
```

is followed by the bitmaps definitions for all the icons which will be used in the browser for the different classes of objects.

The application can define its own icons to represent objects in the browser. Each class definition allows one to specify the names of "big" and "small" icons which are looked up in the table of available icon bitmaps. The same icon is used for both sizes only one of them is defined. If none is defined then a default icon is used. The `>Class` directive refers to icons by name, e.g. "`sm_myicon`".

To create a new icon bitmap one can use the X11 standard bitmap editor, e.g., to get a 20×20 pixel icon called "`small_1d`" one can type: `bitmap small_1d.bm 20x20`. Then the output file `small_1d.bm` containing "`#define small_1d_width 20` ..." simply needs to be inserted into the CDF following the directive `>Icon_bitmaps`, for instance,

```
>Icon_bitmaps
#define sm_myicon_width 18
#define sm_myicon_height 22
static char sm_myicon_bits[] = {
   0xff, 0xff, 0x03, 0x01, 0x00, 0x02, 0x81, 0x07, ...
   0x21, 0x10, 0x02, 0x21, 0x10, 0x02, 0x21, 0x10, ...
   0x41, 0x08, 0x02, 0x81, 0x04, 0x02, 0x01, 0x03, ...
   0x81, 0x04, 0x02, 0x41, 0x08, 0x02, 0x41, 0x08, ...
   0x21, 0x10, 0x02, 0x21, 0x10, 0x02, 0x41, 0x08, ...
   0x01, 0x00, 0x02, 0xff, 0xff, 0x03};
#define big_myicon_width 36
   ...
```

`KUIPC` compiles the bitmaps into the application, i.e., there is no need for separate bitmap files at run-time. Alternatively bitmap files can be specified as resources in `.Xdefaults` overriding CDF bitmaps of the same name.

3.4.5.4 Help Definition

Many "Help" buttons or menu items are automatically created by KUIP/Motif but for some of them the information required depends directly from the application. This information can be written in the CDF following the "`Help_item`" directive with predefined key-words.

```
>Help_item  HELP_EXE
```

This help concerns the application itself. It can be accessed in the "Help" menu of the **"Executive Window"** (KXTERM) or the "Browser Window", for the menu entry "On application-name" (e.g. "On PAW++").

```
>Help_item  HELP_EXE_RESOURCES
```

This help concerns the X Resources specific to the application. It can be accessed in the “Help” menu of the **“Executive Window”** or the ‘Browser Window”, for the menu entry “On application-name Resources” (e.g. “On PAW++ Resources”).

```
>Help_item  HELP_[brclass]
```

It is possible to define such a “help” for each browsable definition, by replacing “[brclass]” with the browsable-class name. The “help” for a specific browsable can be accessed by selecting the last entry of the pop-up menu which is displayed when pressing the <mouse button 3> (the last entry of this menu is always “Help”).

It is also possible to use the CDF directive “`>Help_item`” for your application specific “Help” buttons or menus, in the case you have been involved in direct Motif programming. In that case you can use the KUIP/Motif user callable routine “km_do_help” for the call-back associated to your “help” button, in the following way:

```
extern void km_do_help();

char *help_string = ...  /* key-word for ``Help_item'' */

XtAddCallback (helpB1,XmNactivateCallback,
               (XtCallbackProc)km_do_help,(XtPointer)help_string);
```

This routine simply executes the KUIP command: “/KUIP/HELP help_string” where “help_string” is the key-word which follows a “`>Help_item`” directive.

3.4.5.5 Hooks

If one wants to get involved in Motif programming one is free to do so. KUIP/Motif provides hooks that an application can link its own Motif code in.

The CDF directive

```
>Motif_customize  fall-back-routine  widget-routine
```

defines two routines which are called at the initialization phase.

- the first routine (“fall-back-routine”) has to return the application specific fall-back resources. These user fall-backs are appended at the end of KUIP fall-backs (which can then be overwritten). The return value must be a pointer to a NULL-terminated array of resource specification strings to be used if the application class resource file cannot be opened or read. This is also a good way to change the default resource values without having to edit an application class resource file.
  The routine (C coded) has no parameter and the calling sequence is:

  ```
  static String fall_backs[] = {
   ...
  ```

```
 NULL
};
String *fall_back_routine ()
{
   return fall_backs;
}
```

E.g. in PAW++, this routine is called get_paw_fallbacks:

Example of a "fall-back-routine" (get_paw_fallbacks)

```
static String paw_fallbacks[] = {
 "Paw++*kxtermGeometry:                  650x450+0+0",
 "Paw++*kuipGraphics_shell.geometry:     600x600-0+0",
 "Paw++*kuipBrowser_shell.geometry:      +0+485",
 "Paw++*histoStyle_shell.geometry:       +670+635",

 "Paw++*iconForeground:                  grey40",
 ...
 NULL
};
String *get_paw_fallbacks ()
{
   return paw_fallbacks;
}
```

For a more precise description of the most important KUIP/Motif X resources see section 2.8.4.

– the second routine ("widget-routine") is called by KUIP after creating every top level widget (**"Executive Window"**, any browser instance, graphics window) in order to customize these widgets: e.g. to define and set your own pixmaps for the iconic state of each window, to change the titles, etc.
The routine (C coded) has two input parameters: the name of the window as defined in KUIP, and the widget identifier given by Motif. The KUIP predefined names for the different windows are:

  – "kxterm" for the **"Executive Window"** (or KXTERM),
  – "kuipPanel" for any user-defined panels of commands,
  – "kuipGraphics1", ""kuipGraphics2", ..., for graphical windows managed by HIGZ,
  – "kuipBrowser1", "kuipBrowser2", ..., for any browser instance,
  – "kuipScroll" or "kuipScroll1" for any text "ScrolledWindow" managed by KUIP (e.g. "HELP windows").

The calling sequence and skeleton of this user-written routine is:

"widget-routine" skeleton

```
void widget-routine(name, top)
     char *name;  /* Name of the window as defined in KUIP */
     Widget top;  /* Widget identifier given by Motif      */
{
   if (strcmp(name, "kxterm") == 0) {
      ...
   }
```

```
        if (strcmp(name,"kuipPanel") == 0) {
           ...
        }


        if (strncmp(name, "kuipBrowser", 11) == 0) {
           if (strcmp(name, "kuipBrowser1") == 0) {
               ...
           }
           if (strcmp(name, "kuipBrowser2") == 0) {
               ...
           }
           ...
        }

        if (strncmp(name, "kuipGraphics", 12) == 0) {
           if (strcmp(name, "kuipGraphics1") == 0) {
               ...
           }
           if (strcmp(name, "kuipGraphics2") == 0) {
               ...
           }
           ...
        }

        ...

     }
```

3.4.5.6 Example: the PAW++ CDF Customization Part

PAW++ CDF (extract)

```
>Name PAMDEF

>Motif_customize get_paw_fallbacks%C init_top_level_window%C

>Button show_histoStyle  ' Style Panel '

>Graphics

...

>Class 1d 1d-Histogram big_1d sm_1d
 Plot                           .  . default_action%C
'/Fit...'                       .  'Histo/Fit [this]'
...

>Icon_bitmaps

#define big_1d_width 30
#define big_1d_height 30
static char big_1d_bits[] =
```

```
   0xff, 0xff, 0xff, 0x3f, 0x01, 0x00, 0x00, 0x20, 0x01, 0x00, 0x00, 0x30,
...
   0xa9, 0xaa, 0xaa, 0x3a, 0xfd, 0xff, 0xff, 0x3f, 0xff, 0xff, 0xff, 0x3f;

#define sm_1d_width 20
#define sm_1d_height 20
static char sm_1d_bits[] =
   0xff, 0xff, 0x0f, 0x01, 0x00, 0x08, 0x01, 0x00, 0x0c, 0xa9, 0xaa, 0x0e,
...
   0xf1, 0xff, 0x0d, 0xa9, 0xaa, 0x0e, 0xfd, 0xff, 0x0f, 0xff, 0xff, 0x0f;

...

>Help_item HELP_EXE

>Guidance
       *** Paw++ ***
.
Paw++ is a new and powerful OSF/Motif based Graphical User Interface
to the popular Physics Analysis Workstation (PAW).
...
>Help_item HELP_EXE_RESOURCES
>Guidance
       *** X Resources for Paw++ ***
.
This is a list of the X resources available to Paw++.
Resources control the appearance and behavior of an application.
...
>Help_item HELP_Hbook
>Guidance
       *** Class "Hbook" ***
.
The class "Hbook" allows to browse the file system for Hbook
files. Hbook files are files with the extension ".hbook".
...
```

3.5 Compiling and Linking

The CDF has to be translated by KUIPC into Fortran or C source code which is then passed to the appropriate compiler. The resulting object file is then linked with the rest of the application specific routines and the referenced libraries into an executable module.

Here we want to summarize the system dependencies in these three steps. The KUIP compiler has the following command line syntax:

```
kuipc  [ options ]  [ input-file  [ output-file ] ]
```

In order for the `kuipc` executable to be found on Unix the directory `/cern/pro/bin` must be included in the search path. On VMS there must be a symbol definition such as

$OS/$MACHINE	Fortran compilation	C compilation[a]
UNIX/ALPHA	f77 -c	cc -c
UNIX/APOLLO	ftn *file*.ftn -b *file*.bin	ver bsd4.3 /bin/cc -c -DAPOFTN \ -o *file*.bin *file*.c
UNIX/APOLLO f77	f77 -c	ver bsd4.3 /bin/cc *file*.c
UNIX/CONVEX	fc -c -fi	cc -c -fi
UNIX/CRAY	cft77	cc -c
UNIX/DECS	f77 -c	c89 -c
UNIX/HPUX	f77 -c +ppu +T	cc -c -Ae
UNIX/IBMRT	xlf -c -qextname -qcharlen=1024	cc -c
UNIX/LINUX	f77 -c -Nx800 -Nc200	cc -c -posix
UNIX/NEXT	f77 -c -f -N3 -N9 -N1	cc -c -I/usr/include
UNIX/SGI	f77 -c	cc -c
UNIX/SUN	f77 -c	acc -c
UNIX/SOLARIS	f77 -c	cc -c -DSolaris2
VM/IBM	vfort *file*	cc *file* (alias(*file*) define(IBM370)
VMS/ALPHA	fortran *file*	cc/common=extern/prefix=all *file*
VMS/VAX	fortran *file*	cc *file*
MSDOS/IBMPC	fc -c	xgcc -c

[a] Unless otherwise noted the compile commands are supposed to end with the source file name *file*`.f` or *file*`.c`, respectively. The commands given here contain only the minimum options necessary for compiling the KUIPC generated file. The compilation of other source files may require additional options for debugging, optimization level, header file directories, etc.

Table 3.1: Compilation commands

```
KUIPC == "$CERN:[PRO.EXE]KUIPC"
```

If no input file is given on the command line KUIPC prompts for the input and output file specifications. The input file has the default extension "`.cdf`".

If only the input file is specified on the command line the output file will have the same name but a different extension depending on the mode. On the other hand the output mode, i.e. whether Fortran or C code is generated, can also depend on the output file extension:

- The option "`-c`" forces C mode and the output file extension defaults to "`.c`".
- The option "`-f`" forces Fortran mode and the output file extension defaults to "`.fortran`" on VM/CMS, "`.for`" on VMS, or "`.f`" on any other system.
- Otherwise C mode is selected if the output file specification contains the explicit extension "`.c`".
- Otherwise the output mode is Fortran, and the default file extensions stated above apply.

Another option is `-split` which instructs KUIPC to write the generated code into separate files as given by the `>Name` directive. This is especially useful for the C output mode if the CDF contains information which should be loaded only conditionally.

	Examples for invoking KUIPC
`kuipc file`	translate `file.cdf` into `file.f`
`kuipc file.txt file.c`	translate `file.txt` into `file.c`
`kuipc -split -c file`	translate each `>Name` part of `file.cdf` into a separate C~file

The output file generated by KUIPC has then to be compiled into an object module. The necessary compilation options are listed in table 3.1.

Using the C output mode of KUIPC is preferable over generating Fortran source code. The Fortran mode is intended for backward compatibility in order not to force the use of the C compiler in an otherwise Fortran-only environment. New features such as action routines written in C and KUIP/Motif specific CDF directives are only supported in C output mode.

3.6 Various Hints specific to KUIP/Motif

3.6.1 C-callable Interface to the Panel Interface

A C-callable interface for the complete `PANEL` interface (panels and palettes) built-in inside KUIP/Motif (see 2.8.3) is accessible to the application programmer. This allows him to set up some predefined panels without having to transport independent macro files together with the application executable, and having to refer them in the logon macro file. This is the case, for example, of the graphical editor “Ged” which makes an intensive use of graphical and alphanumerics “programmer-defined” panels.

The C callable entries are:

```
void km_panel_reset( const char *name )
```

Parameter Description:

`name` panel name (can be NULL; it is mandatory if several panels use the same button label/alias for different commands; then one has to give different names to each panels).

Reset panel in memory (<> “`panel 0 r ...`”).

```
void km_panel_key( int row, int col,  char *command,
                   char *alias_label, char *pixmap )
```

Parameter Description:

`row`	row number
`col`	column number
`command`	command (or list of commands) to be executed
`alias_label`	alias name for command (or NULL)
`pixmap`	pixmap representation for button label (or NULL)

Define key (<> “`panel x.y ...`”).

```
void km_panel_display( char *title, char *geometry )
```

Parameter Description:

`title` panel title
`geometry` panel geometry (*w*x*h*+*x*+*y*)

Display panel (<> "`panel 0 d ...`").

```
void km_panel_close( char *title )
```

Parameter Description:

`title` panel title

Close panel with given title (<> "`panel 0 c ...`").

```
void km_icon( char *icon, char *fname )
```

Parameter Description:

`icon` icon name
`fname` filename (output of the X11 utility bitmaps).

Store icon data from a bitmap file (<> "`motif/icon ...`").

```
void km_palette( char *title, char *geometry )
```

Parameter Description:

`title` palette title
`geometry` palette geometry (wxh+x+y)

Open or close a "multi-panel" (palette) widget (<> "`motif/multi_panel ...`").

The routine (written by the application programmer) which contains all the panels definition has to be called via the "Motif_customize" mechanism (described is section 3.4.5.5) in the so-called "widget-routine" (its has to be called before entering the Motif main loop, but after initialization).

E.g.:

```
>Motif_customize . init_top_level_window%C
```

The following is an illustration of how to use these routines. On the left side you have the KUIP macro file for a panel definition, and on the right side you have the equivalent coded in C. The C routine `my_panel_definition` is automatically called by KUIP, just after the creation of the **"Executive Window"** (kxterm), thanks to the CDF directive "Motif_customize". (See the code of the "widget-routine" `init_top_level_window` at the end of the "C Interface" side).

```
**************          |     /***************/
* KUIP Macro *          |     /* C Interface */
**************          |     /***************/

                        |     #include <X11/Intrinsic.h>

                        |     extern void km_icon();
                        |     extern void km_panel_reset();
                        |     extern void km_panel_display();
                        |     extern void km_panel_key();

                        |     my_panel_definition()
                        |     {
*
* Icon bitmaps
*
/motif/icon m1 mk1.bm   |       km_icon ("m1", "/user/cremel/ktest/mk1.bm");
/motif/icon m2 mk2.bm   |       km_icon ("m2", "/user/cremel/ktest/mk2.bm");
/motif/icon m3 mk3.bm   |       km_icon ("m3", "/user/cremel/ktest/mk3.bm");
/motif/icon m4 mk4.bm   |       km_icon ("m4", "/user/cremel/ktest/mk4.bm");
/motif/icon m5 mk5.bm   |       km_icon ("m5", "/user/cremel/ktest/mk5.bm");

*
* Panel keys definition
*
panel 0                 |       km_panel_reset( NULL );
panel 2.01 null         |       km_panel_key (2, 1, "null", NULL, NULL);
panel 2.02 tex_1        |       km_panel_key (2, 2, "tex_1", NULL, NULL);
panel 3.01 '/example/general kuip.tex tex 1' 'tex_1' m1
                        |       km_panel_key (3, 1, ...
                          ..."/example/general kuip.tex tex 1", "tex_1", "m1");
panel 3.02 '/example/general kuip.tex tex 2' 'tex_2' m2
                        |       km_panel_key (3, 2, ...
                          ..."example/general kuip.tex tex 2", "tex_2", "m2");
panel 3.03 '/example/general kuip.tex tex 3' . m3
                        |       km_panel_key (3, 3, ...
                          ..."example/general kuip.tex tex 3", NULL, "m3");
panel 3.04 '/example/general kuip.tex tex 4' . m4
                        |       km_panel_key (3, 4, ...
                          ..."example/general kuip.tex tex 4", NULL, "m4");
panel 4.01 ' ' . m5     |       km_panel_key (4, 1, " ", NULL, "m5");
panel 4.02 'tex_5' . m5 |       km_panel_key (4, 2, "tex_5", NULL, "m5");
panel 5.01 '/example/general kuip.tex tex 6' . m6
                        |       km_panel_key (5, 1, ...
                          ..."example/general kuip.tex tex 6", NULL, "m6");
panel 5.02 '/example/general kuip.tex tex 6' . big_menu
                        |       km_panel_key (5, 2, ...
                          ..."example/general kuip.tex tex 6",NULL,"big_menu");
panel 6.01 '/example/general kuip.tex tex 7' 'tex_7'
                        |       km_panel_key (6, 1, ...
                          ..."example/general kuip.tex tex 7", "tex_7",NULL);
```

```
panel 6.02 '/example/general kuip.tex tex 7' 'tex_7' m1
                         |      km_panel_key (6, 2, ...
                           ..."example/general kuip.tex tex 7", "tex_7", "m1");

* Open a palette (multi_panel)
multi_panel 'Marker Palette' '300x1000-0+0'
                         |      km_palette ("Marker Palette", "300x1000-0+0");

* Display panel(s)
panel 0 d 'Marker Types' 300x300+500+500
                         |      km_panel_display ("Marker Types", ...
                           ..."300x300+500+500");

* Close current palette
multi_panel 'end'        |      km_palette ("end", NULL);
                         |    }

                         |    void init_top_level_window(name, top)
                         |        char *name;
                         |        Widget top;
                         |    {
                         |      if (strcmp(name,"kxterm") == 0) {
                         |        my_panel_definition();
                         |      }
                         |    }
```

3.6.2 C/Fortran-callable Interface to the Browser Interface

The C-callable routine *km_browser_set* is accessible to the application programmer to initialize the main browser window in a pre-defined state as soon the application is starting. A Fortran entry *KMBRSE* which calls this routine is also provided.

```
void km_browser_set( const char *browsable, const char *path )
```

Parameter Description:

`browsable`	name of the file (browsable) to be selected in the browser.
`path`	(can be NULL) pathname to be used for that browsable if it is not the default one.

E.g. By calling in a C routine,

```
km_browser_set ("Files", "/user/paw/demo")
```

or in Fortran,

```
KMBRSET ("Files", "/user/paw/demo")
```

before calling "KUWHAM" (Motif main loop) the main browser window will display at start up the list of files in the directory "/user/paw/demo".

3.6.3 User-Defined Single Selection Lists

It is possible in KUIP/Motif to set up a command which displays a list of various items or choices (instead of the usual command argument panel with the list of arguments to be filled). This is, for instance, the behavior of the command "HELP" in KUIP/Motif: if you type "help" without any argument a selection box with all possible help menus or items is displayed and the user can select one (or just cancel the execution) and get the appropriate help information.

Note that the argument type "list" does not exist in the basic KUIP and we thought that the most frequent case is probably the one mentioned above, i.e. a command which displays a predefined list (or set up at run time). In such a case one should get access to the "Selection Box" widget in a so-called Motif interface.

To implement such a command the application-programmer has to define in the CDF (with the other application defined commands) a new command without any argument and the action routine written in C. E.g.:

```
>Menu EXAMPLE
...
>Command LIST
>Guidance
This is just an example of a command which
displays a list of items.
(See action routine ktactc.c)
>Action ktactc%C
```

We provide in KUIP/Motif two user-callable C routines (`km_list_data` and `km_show_list` which make the implementation of this "list display" quite easy (no Motif code is involved). These routines have to be called directly in the code of the action routine. The part which concerns the filling of the list and the action to execute after a selection has been done (<OK> button pressed) is (obviously) application dependent.

Description of these two user-callable C routines:

```
void km_list_data( char *list_label, char *selection_label,
                   char *help_text,  int call_back(char* selection) )
```

Parameter Description:

`list_label`	label (prompt) written at the beginning of the list
`selection_label`	label (prompt) written before the selection
`help_text`	help message accessed through help button
`call_back`	user defined routine called when the OK is pressed. The value selected by the user is passed as argument when the routine is called.

Fill the application data for a user-defined list. All parameters are application dependent.

```
void km_show_list( char **items )
```

Parameter Description:

`items` list of items (choices) in the list.

Displays the list.

The following is a skeleton (some parts have to be filled according to the application) for the application routine `ktactc.c` for the new command "LIST" described above:

```
/* Listing of file ktactc.c */

#ifndef NULL
#define NULL 0
#endif

/****************************************************************/
/*                                                              */

/* Action routine for command "/EXAMPLE/LIST"                   */
/* (Example of how to display a "user defined" list             */
/*  with KUIP/Motif).                                           */
/*                                                              */
/****************************************************************/

ktactc()
{
  int ktactc_OK ();
  int i;

  static char *help_text = "You can write there \n\
some help text for \"My List\" ...";

  /* List of all items (can be filled dynamically) */
  char **list_item = (char **) malloc ( sizeof (char *) );

  /* Fill list data: */
  km_list_data
     ("My List Label", "My List Selection", help_text, ktactc_OK);

  /* Fill list of items (with 10 arbitrary values) ...
   * ... N.B. Write your application defined code here ... i
   */
  for (i = 0; i < 10; i++) {
       list_item = (char **) realloc( (char*)list_item,
                                     (i+1) * sizeof (char *) );
       list_item[i] = (char *) malloc (10);
       sprintf (list_item[i], "item %d", i+1);
  }
  list_item = (char **) realloc( (char*)list_item, (i+1) * sizeof (char *) );
  list_item[i] = NULL;
```

```
  /*
   * Display list and wait for user action ...
   * ... N.B. If user action is "OK" you go into
   * the "OK call-back" defined by km_list_data (ktactc_OK)
   * - See example for code of ktactc_OK above _
   */
  km_show_list (list_item);

  for (i = 0; i < 10; i++) free (list_item[i]);
  free (list_item);
}

/*
 * Routine name (ktactc_OK) is defined by KM_list_data.
 * This routine is automatically called when pressing the "OK" button.
 *      char *val (input) : value issued from the user selection
 *
 */
int ktactc_OK (val)
    char *val;
{
    printf ("*** display_list_OK : (ktactc_OK)\n");
    printf ("    value selected in the list %s.\n", val);

    return 0;
}
```

Picture 3.5 shows the output from KUIP/Motif when executing the command “/EXAMPLE/LIST” described above.

Figure 3.5: User-defined List in a Command

3.6.4 User-Defined File Selection Boxes

It is also possible with KUIP/Motif to set up a command which displays a "FileSelectionBox" (conform to the Motif terminology) containing:

- a “Filter” entry to select only files with a certain pattern,
- a “Directories” entry to select the directory applied to the “Filter” (this can also be done by editing the “Filter” entry manually),
- a “Files” entry, with the list of all files corresponding to the “Filter” selected.

This is a very convenient way to select a file name, and being able to scan the directory tree.

To implement such a command the application-programmer has to define in the CDF (with the other application defined commands) a new command without any argument and the action routine written in C. E.g.:

```
>Menu EXAMPLE
...
>Command SELECT_FILE
>Guidance
This is just an example of a command which
displays a file list selection box.
(See action routine ktactcf.c)
>Action ktactcf%C
```

We provide the user-callable a C routine (`km_show_filSel`) which makes the implementation of this “FileSelectionBox” quite easy (no Motif code is involved). It has to be called directly in the code of the action routine. The part which concerns the data setting and the action to execute after a selection has been done (<OK> button pressed) is (obviously) application dependent.

N.B. You get automatically to this "FileSelectionBox" by defining a parameter type “FILE” (extension to the type “`C`” (character) in the CDF). But this C user-callable routine gives the application programmer the possibility to define commands that directly display a "FileSelectionBox" (without producing the usual “Command Argument Panel”) and also to fill the data (filter, default value) at run time.

Description:

```
void km_show_filSel( char *title, char *dir, char *def, char *help,
                     int call_back(char* selection) )
```

All parameters (input) in this routine are application dependent.

Parameter Description:

`title`	title of the FileSelectionBox
`dir`	directory (for filter)
`def`	default file name
`help`	help text (accessed through help button)
`call_back`	user defined routine called when the “OK” button is pressed. The value selected by the user is passed as argument when the routine is called.

The following is a skeleton (some parts have to be filled according to the application) for the application routine `ktactcf` for the new command `SELECT_FILE` described above:

```
/* ktactcf.c */

#include <stdio.h>

extern void km_show_filSel( char *title, char *dir, char *def, char *help,
                            int callback(char* selection) );


/*
 * This routine is automatically called when pressing the "OK" button.
 *      char *val (input) : value issued from the user selection
 */
int ktactcf_OK( char *val )
{
    printf( "*** fileSelection OK : (ktactcf_OK)\n" );
    printf( "    File selected is %s.\n", val );

    return 0;
}


/*
 * Action routine for command "SELECT_FILE".
 * (Example of how to display a "user defined" file selection
 *  box with KUIP/Motif).
 */
int ktactcf()
{


  static char *help_text = "You can write there \n\
some help text for \"My FileSelectionBox\" ...";

    km_show_filSel( "My FileSelectionBox",
                    "/user/cremel/pdemo/*.dat", "hrztest.dat",
                    help_text, ktactcf_OK );
    return 0;
}
```

Picture 3.6 shows the output from KUIP/Motif when executing the command `SELECT_FILE` described above.

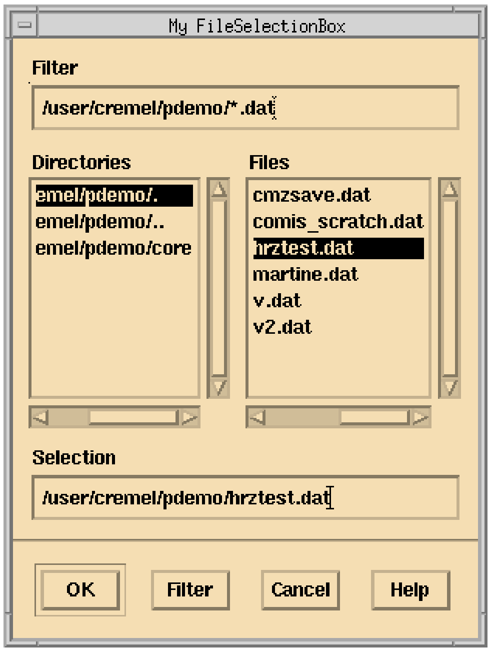


Figure 3.6: User-defined File Selection Box in a Command

3.7 Some Examples to Start With

3.7.1 Example 1: Basic Example (No Graphics)

This first example is a very simple one in order to show

1 how the different types of parameters handled by KUIP are interpreted and visualized in the Motif interface ("Command Argument Panel").

2 what you automatically get with KUIP/Motif.

3.7.1.1 The "Command Argument Panel" for a Very General Command

The following CDF defines a new menu "Example" and a new command "General" with 6 input parameters. The first 2 parameters are mandatory and the others are optional.

The action routine to be called at command execution is named KTACT.

Example 1 — CDF (Command Definition File) — ktcdf.cdf

```
>Name KTDEF

>Menu Example

>Command General
>Parameters
FILE 'File name' C D='kuip.tex'
OPTION 'Text formatting system' C D='TEX'
-      plain text : plain text format
-LATEX LaTeX format (encapsulated)
```

```
-TEX   LaTeX format (without header)
+
LINE 'Number of lines' I D=10
HEIGHT 'Page height (cm)' R D=28.5
LENGTH 'Length of space between sections (cm)' R D=2.5 R=-2.:5.0
PAGE  'Page number' I D=1 R=1:99
>Guidance
This is just an example of a command with different parameter
types.
>Action KTACT
```

The “Command Argument Panel” which is automatically generated by KUIP/Motif for this very general command “/EXAMPLE/GENERAL” is show in figure 3.7.

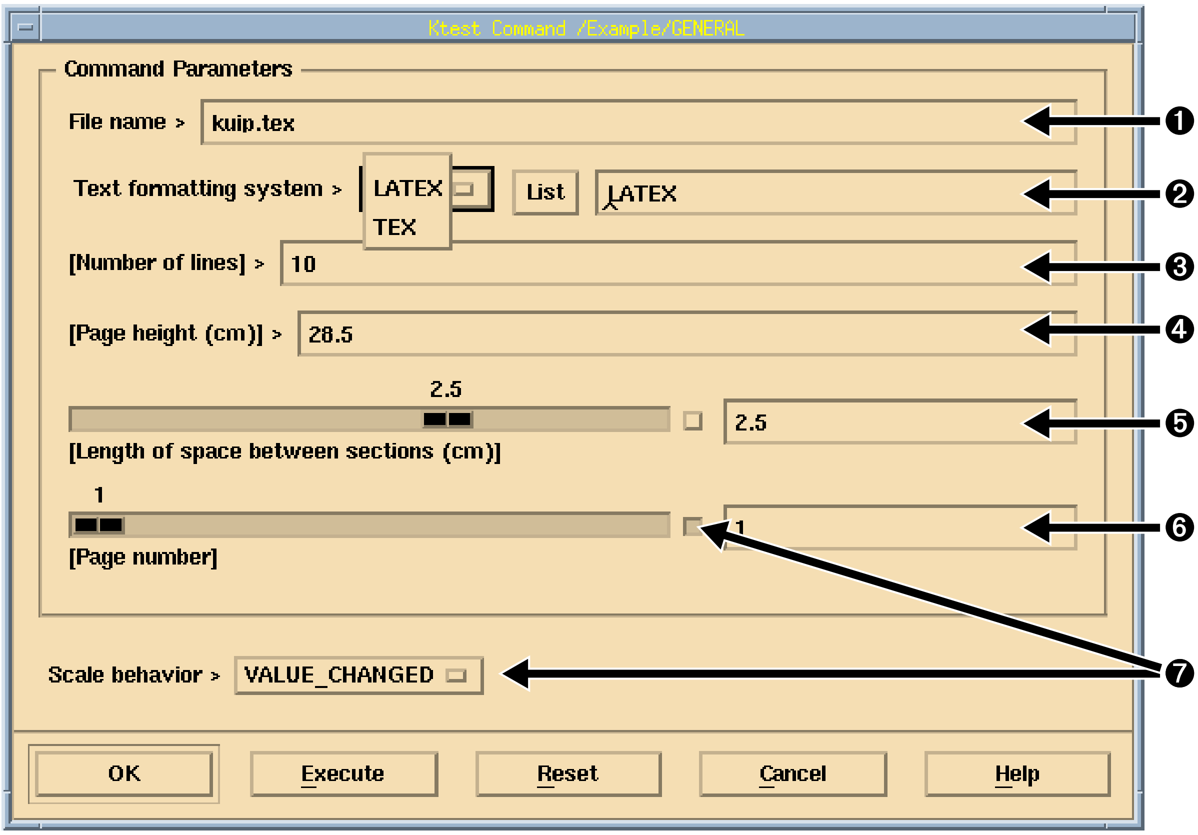


Figure 3.7: The “Command Argument Panel” for command EXAMPLE/GENERAL (Example 1)

❶ Parameter 1 (FILE): string input for ’File name’ (default is ’kuip.tex’). This is represented by a “Text” widget.

❷ Parameter 2 (OPTION): string input for ’Text formatting system’ with a predefined range of possible values. This is represented by an “Option Menu” widget + access to a “List” widget with the complete description of the possible options + string input (where the user can write any value he wants). The default value is ’TEX’ and possible values are ’ ’, ’LATEX’ and ’TEX’.

❸ Parameter 3 (LINE): integer value for the ’Number of lines’. This is represented by a “Text” widget but an integer value is required (default is 10).

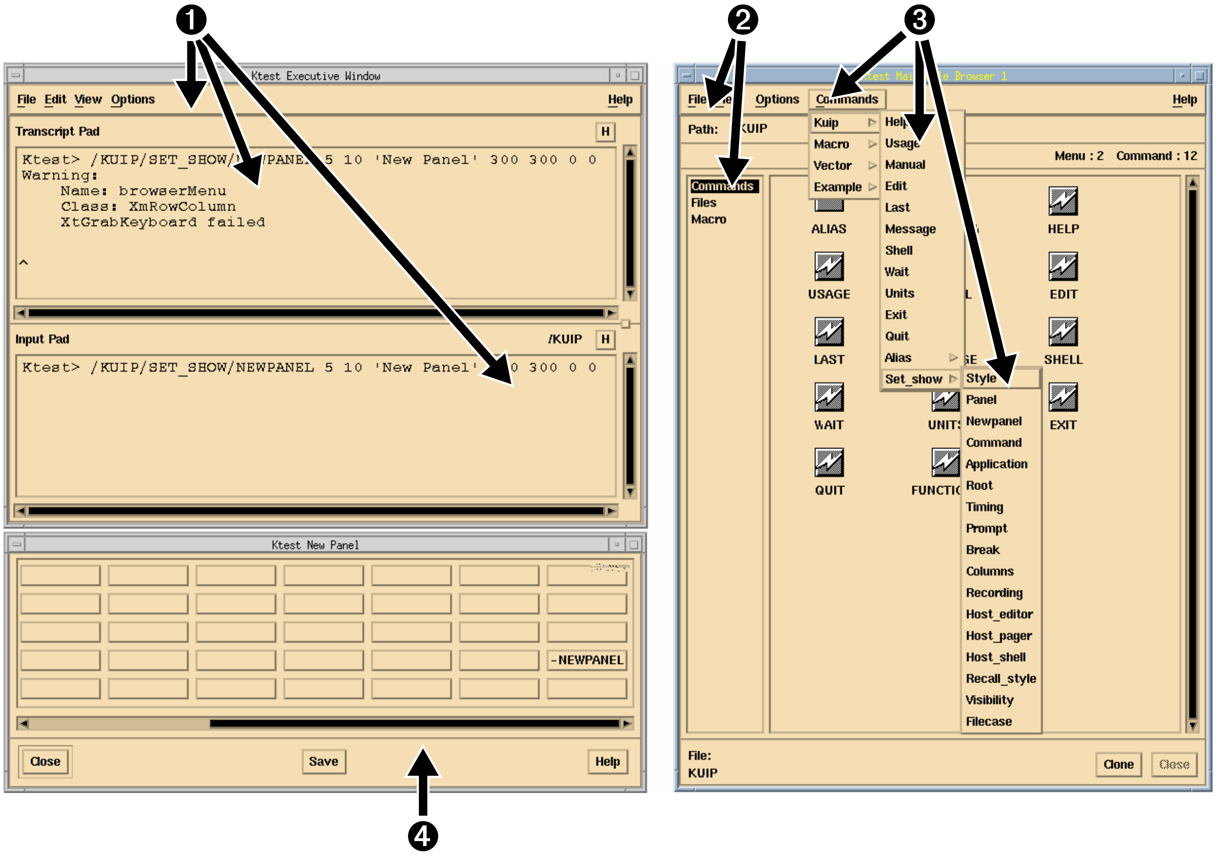
❹ Parameter 4: real value for the ’Page height (cm)’. This is represented by a “Text” widget but a real value is required (default is 28.5).

❺ Parameter 5 (LENGTH): real value for the ’Length of space between sections (cm)’. Possible values have to be within the range [-2.0...5.0]. This is represented by a “Scale” widget (for reals) + string input (where the user can write any value he wants). The default value is 2.5.

❻ Parameter 6 (PAGE): integer scale for the ’Page number’ within the range [1...99]. This is represented by a “Scale” widget (for integers) + string input (where the user can write any value he wants). The default value is 1.

❼ This is an option menu which gives the user the possibility to change the behavior for scales (when real or integer ranges of values are given for parameters). By default the scale behavior is set to “VALUE_CHANGED” but it can be set to “DRAG” which means that the value of the parameter corresponding to the scale is modified as soon as the user “drag” the scale (and not only when releasing the mouse button on a certain value). When the little toggle button next to the scale is highlighted the command execution is effective as soon as the scale is modified without the user has to press the “OK” or “Execute” button and whatever the scale behavior setting is (“VALUE_CHANGED” or ”DRAG”). This allows the user to see dynamically the effect of some parameters setting: e.g. in PAW++ for the command “HISTOGRAM/OPERATIONS/SMOOTH” one can see dynamically the changes in the smoothing algorithm according to the value of the “sensitivity” or the “smoothing” parameters.

N.B: For parameters 2, 5, and 6 the user has several possibilities to set the value: option menu, list or scale selection, or directly string input. In any case it is the value which is written into the small text area for string input that is taken for the command execution. (The option menu, list and scale selection modify automatically the content of this text area).



❶ The **"Executive Window"** with its menu-bar, the **Transcript Pad** and the **Input Pad** (where users can type commands).

❷ The **"Main Browser"** with its menu-bar, and the predefined KUIP browsables for the "Commands", "Files" and "Macro".

❸ Pulldown menu access to all the commands defined by KUIP and the application.

❹ The possibility to build panels of commands.

Figure 3.8: What Do You Get? (Example 1)

3.7.1.2 Building the Example: What Do you Get?

The CDF file has to be compiled by the KUIPC compiler into Fortran or C code (according to the file type extension "`.f`" or "`.c`" given to the output file). With KUIP/Motif we require the generation of C code, as all the CDF extension specific to Motif (section 3.4) are only interpreted in the C output mode of KUIPC. The command to be given on any system is

```
kuipc ktcdf.cdf ktcdf.c
```

which will generate the file `ktcdf.c` (to be compiled by the C compiler) from the input file `ktcdf.cdf`.

The following is the application main program in Fortran:

Example 1 - Main Program - “ktest_main.f”

```
      PROGRAM KTEST
*
* Basic KUIP Application with MOTIF / NOT linked to HIGZ
*
      COMMON/PAWC/PAW(500000)
      CALL INITC
*
* Initialize PAW
*
      CALL MZEBRA(-3)
      CALL MZPAW(500000,' ')
*
* Initialize KUIP with NWORDS words as minimum division size
*
      NWORDS=20000
      CALL KUINIT(NWORDS)
*
* Create user command structure from definition file
*
      CALL VECDEF
      CALL KTDEF          ! Command(s) definition part
*                         ! (code generated from the CDF compilation)
*
* Gives access to KUIP browsers for commands, files and macros
*
      CALL KUIDFM
*
* Execute some KUIP Initialization Commands
*
      CALL KUEXEC('PROMPT ''KTEST >''')
      CALL KUEXEC('LAST 0')
*
* Give control to MOTIF
*
      CALL KUWHAM ('Ktest')
*
      END
```

We have set the class name of this basic example to “`Ktest`” (`CALL KUWHAM('Ktest')`): this means that the X resources for the application have to be referred with :

`Ktest*`*<resource_name>* : *<resource_value>*

In the action routine (`KTACT`), coded in Fortran, we just retrieve and print the value of all parameters:

Example 1 - Action Routine KTACT - (“ktact.f”)

```
      SUBROUTINE KTACT
*
      CHARACTER*32 CHPATH, CVAL, CHOPT

      DATA LUN/93/
*
```

```
* Retrieve command name and number of parameters
*
      CALL KUPATL (CHPATH,NPAR)
*
      IF (CHPATH.EQ.'GENERAL') THEN
*
* Retrieve parameter values
*
           CALL KUGETC (CVAL, IL1)
           CALL KUGETC (CHOPT, IL2)
           CALL KUGETI (ILINE)
           CALL KUGETR (RHEIGHT)
           CALL KUGETR (RLENGTH)
           CALL KUGETI (NBP)
           print *, '===> Command GENERAL:'
           print *, '     File : ', CVAL(1:IL1)
           print *, '     Option : ', CHOPT(1:IL2)
           print *, '     Number of lines: ',ILINE
           print *, '     Page height: ', RHEIGHT
           print *, '     Length of space between sections:', RLENGTH
           print *, '     Page number: ', NBP
      ENDIF
*
 99   RETURN
      END
```

The link command for `ktest` is:

```
f77 -o ktest ktest_main.f ktcdf.c ktact.f `cernlib -G Motif`
```

Fig. 3.8 back on page 130 shows what you get for this very basic example.

3.7.2 Example 2: Application with a Graphical Window Managed by HIGZ

This second example shows how to build an application which opens a graphical window managed by HIGZ and how it is possible to obtain "direct object manipulation" in this graphical window by defining some specific "classes" of objects inside the CDF.

The CDF is divided into 2 parts:

- 1.`>Name KTDEF`
  it contains the definition for a new menu "Graphics" with one command "Frame_box" to draw a frame box with axis.
  The action routine to be called at command execution is named KTACTG.
- 2.`>Name KTDEFMG`
  it contains specific definition for the KUIP/Motif interface:
  - The directive "`>Graphics`" is mandatory for an application which requires one (or more) graphical window(s) managed by HIGZ.
  - 3 classes of object are defined ("win", "x-axis" and "y-axis") with their specific menu of actions. These menus apply to objects which are identified in the HIGZ graphics window

(2nd set of menu separated by a blank line starting with "+"). (See section 3.4.3 for more details).

Example 2 — CDF (Command Definition File) — ktcdfg.cdf

```
>Name KTDEF

>Menu Graphics

>Command FRAME_BOX
>Parameters
+
XMIN  'Low range in X'  R D=0. R=0:1000
XMAX  'High range in X' R D=100. R=0:1000
YMIN  'Low range in Y'  R D=0. R=0:1000
YMAX  'High range in Y' R D=100. R=0:1000
CHOPT 'Options'         C D=' '
-  frame box : Draw a frame box only
-S scale : Redefine the scale for the current zone
-A NO axis : Axis labels and tick marks are not drawn
-B NO Box : The box is not drawn
>Guidance
Draw a frame box.
If XMIN, XMAX, etc. are given, draw a frame box
with the window coordinates set to XMIN, XMAX, YMIN, YMAX. Axis
labels and tick marks are drawn by default.
>Action KTACTG

>Name KTDEFMG

>Graphics

>Class win 'Graphics Window'
+
 Plot                              . 'Picture/Plot'
'/Do PostScript...'                . '-Picture/Print test.ps'
'Do Encapsulated PostScript...'    . '-Picture/Print test.eps'
'Do LaTex...'                      . '-Picture/Print test.tex'
'Print'                            . 'Picture/Print'
'/Open New Window'                 . 'Work [this] OA'
'Close Window'                     . 'Work [this] C'
'Activate Window'                  . 'Work [this] A'
'Deactivate Window'                . 'Work [this] D'

>Class x-axis 'X Axis'
+
'Logarithmic'    . 'OPTION LOGX'
'Linear'         . 'OPTION LINX'
'Color'          . 'SET XCOL'
'Character Font' . 'SET VFON'
```

```
>Class y-axis 'Y Axis'
+
'Logarithmic'    . 'OPTION LOGX'
'Linear'         . 'OPTION LINX'
'Color'          . 'SET XCOL'
'Character Font' . 'SET VFON'
```

N.B. The object class "win" is predefined inside HIGZ itself (see the HIGZ documentation for routine IGOBJ) in order to refer the graphics window when no other object with a higher level has been identified. This is a very convenient facility for defining a popup menu associated to a graphics window managed by HIGZ: the only thing to do is to describe this menu of actions in the CDF following the directive ``>Class win ...'' (as done in our example).

The following is the application main program in Fortran:

Example 2 - Main Program - "ktestg_main.f"

```
      PROGRAM KTEST
*
* Basic KUIP/Motif Application with HIGZ graphics window
*
      PARAMETER (NWHIGZ=10000)
      COMMON/PAWC/PAW(500000)
*
      EXTERNAL      IGTERM
      CALL INITC
*
* Initialize PAW
*
      CALL MZEBRA(-3)
      CALL MZPAW(500000,' ')
*
* Initialize KUIP with NWORDS words as minimum division size
*
      NWORDS=20000
      CALL KUINIT(NWORDS)
*
* Create user command structure from definition file (CDF)
*
      CALL VECDEF
      CALL KTDEF          ! Command(s) definition part
*                         ! (code generated from the CDF compilation)
*
* Gives access to KUIP browsers for commands, files and macros
*
      CALL KUIDFM
*
* Special KUIP initialization for using Motif with HIGZ
*
      CALL KTDEFMG        ! Graphics and class(es) definition
*                         ! (code generated from the CDF compilation)
```

```
      CALL KUINIM('Ktestg')
*
* Initialize HIGZ
*
      CALL IGINIT(NWHIGZ)
      CALL KUGRFL(IGTERM)   ! flush the graphics output after each command
*
* Initialize HPLOT
*
      IWK=999
      CALL HPLINT(IWK)
      CALL IGSA(0)
*
* Execute some KUIP Initialization Commands
*
      CALL KUEXEC('PROMPT ''KTEST >''')
      CALL KUEXEC('LAST 0')
*
* Give control to MOTIF
*
      CALL KUWHAM ('Ktestg')
*
      END
```

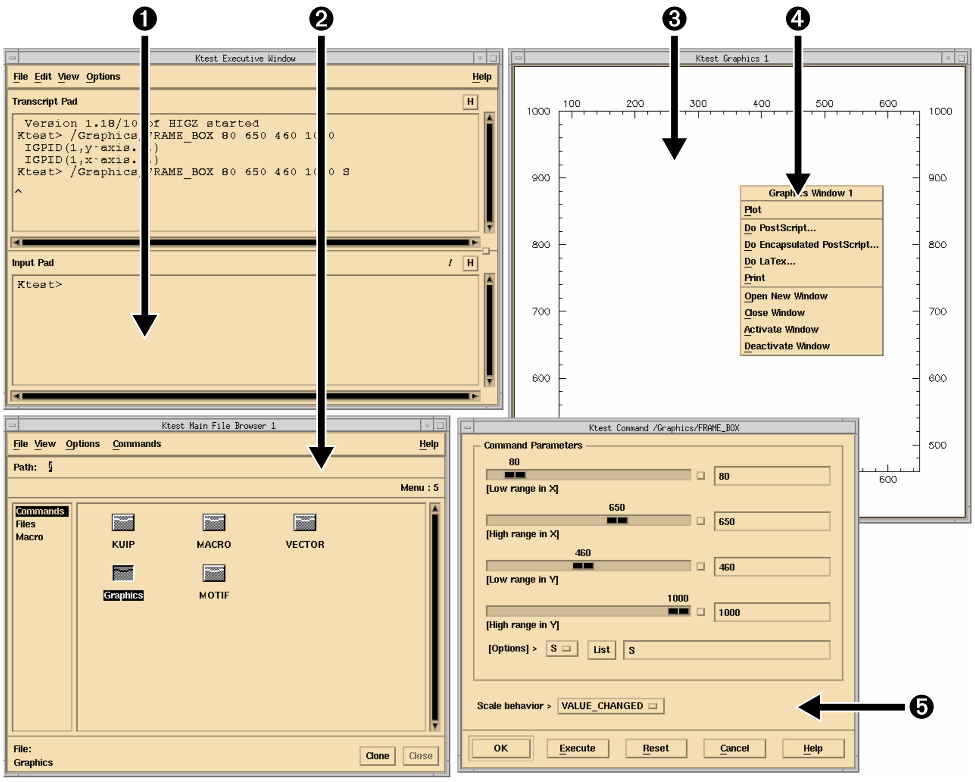
The class name (for X resources setting) of this example is set to "Ktestg" (CALL KUINIM ('Ktestg') and CALL KUWHAM ('Ktestg')).

The following is the action routine (KTACTG), coded in Fortran, for the graphics command /GRAPHICS/FRAME_BOX defined in the CDF. It calls the HPLOT routine "HPLFRA" in order to draw a frame-box in a window opened by HIGZ (axis depend on the parameter values).

Example 2 - Action Routine KTACTG - ("ktactg.f")

```
      SUBROUTINE KTACTG
*
********************************************************************************
*
* Execution routine for command '/GRAPHICS/NULL'
*
********************************************************************************
      CHARACTER*32 CHPATH
      CHARACTER*4 CHOPT
*
* Retrieve command name and number of parameters
*
      CALL KUPATL(CHPATH,NPAR)
*
* Retrieve parameter values and draw box
*
      IF(NPAR.EQ.0)THEN
         CALL HPLNUL
      ELSE
         CALL KUGETR(XMIN)
```

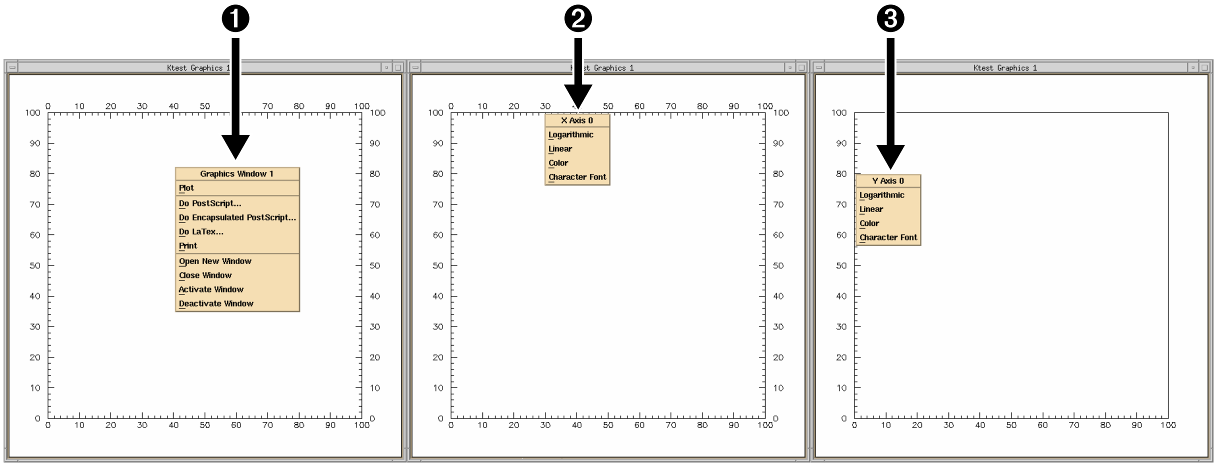
```
           CALL KUGETR(XMAX)
           CALL KUGETR(YMIN)
           CALL KUGETR(YMAX)
           CALL KUGETC(CHOPT,NCH)
*          Create a new picture if necessary
           IF(IZRPIP('PICT00').NE.0)CALL IZPICT('PICT00','S')
           CALL IZPICT('PICT00','M')
           CALL HPLFRA(XMIN,XMAX,YMIN,YMAX,CHOPT)
         ENDIF
*
999      END
```



❶ the **“Executive Window”**,
❷ the **“Main Browser”**,
❸ the HIGZ Graphical Window,
❹ pop-up menu of actions associated to the graphical window (object class “win”),
❺ “Command Argument Panel” associated to the command /GRAPHICS/FRAME_BOX.

Figure 3.9: What Do You Get? (Example 2)

N.B. For object identification inside the graphical window it is necessary to create a HIGZ picture in memory. This is done in our action routine by calling “IZPICT”. The HPLOT routine “HPLFRA” does call internally the HIGZ routine IGPID in order to put the objects “x-axis” and “y-axis” into the HIGZ data structure (see the HIGZ documentation for more information on this routine). This is mandatory for



❶ no specific object has been identified. The pop-up menu corresponding to the predefined class "win" (and described in the CDF) is displayed.

❷ the object "x-axis" has been identified and its corresponding menu (as described in the CDF) is displayed.

❸ the object "y-axis" has been identified and its corresponding menu (as described in the CDF) is displayed.

Figure 3.10: Graphical Object Identification (Example 2)

enabling the "graphical object picking" mechanism at running time.

The result is show in Fig. 3.9. The link command for `ktestg` is:

```
f77 -o ktestg ktestg_main.f ktcdfg.c ktactg.f `cernlib -G Motif graflib`
```

Figure 3.10 shows the graphical window in three different states: the user has pressed the <mouse button 3> in order to access the menu of actions specific to the objects which are identified by HIGZ through the "IGOBJ" and "IGPID" mechanism (described in the HIGZ manual).

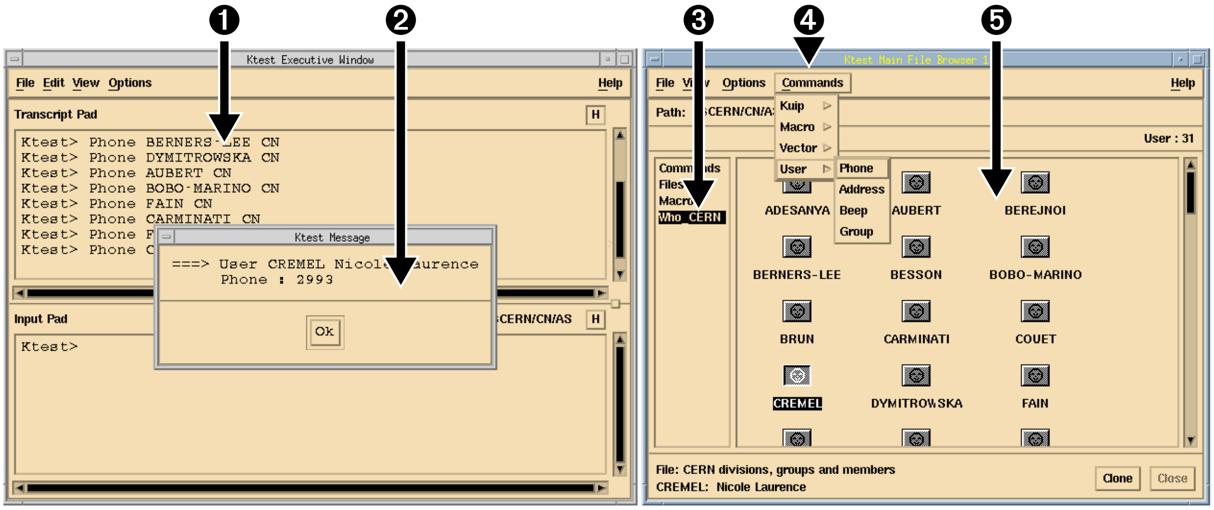
3.7.3 Example 3: How to Build a new Browsable (Who_CERN) ?

The objective of this third and last example is to show what the application programmer has to do in order to create a new browsable class of objects. We have taken a very easy-to-understand example by building a browsable "Who_CERN" which gives access to the CERN hierarchical structure with the divisions, the groups inside each division, and finally all the CERN members inside each group. We have defined a few new commands "Phone", "Address", "Beep", ..., which display the phone number, the address or the beep number of a particular CERN member.

The CDF is divided into two parts:

`>Name KTDEF` containing the definition for a new menu "User" with several commands: "Phone", "Address", "Beep", and "Group". These commands have 2 parameters: the "User Name" (character string) is mandatory, and the "Division" (character string) is optional. The action routine to be called at command execution is named `KTACTB`.

`>Name KTDEFMB` containing specific definitions for the KUIP/Motif browser interface:



❶ the **“Executive Window”** (**Input Pad**).
❷ Output from the “Phone” command execution (message displayed with the KUIP subroutine “KUMESS”).
❸ the browsable entry “Who_CERN” is selected.
❹ Pull-down menu “Commands” with the complete tree command structure.
❺ the **“Main Browser”** (**“Object window”**) for the browsable “Who_CERN”.

Figure 3.11: What Do You Get? (Example 3)

- the browsable “Who_CERN” is defined with its mandatory “scan-objects” routine (`KTSOBJ` in Fortran) and optional “scan-browsable” routine (`ktsbro` in C).
- three classes of objects are defined (“/Division”, “/Group” and “Usr”) with their specific menu of actions. These menus apply to objects which are identified in the KUIP browser(s) (see section 3.4.3 for more details). “/Division” and “/Group” are special classes for subdirectories (first menu item is always “List”).
- Several icon bitmaps (following the directive `>Icon_bitmaps`) for graphical representation inside the browser are defined.

Example 3 — CDF (Command Definition File) — ktcdfb.cdf

```
>Name KTDEF

>Menu USER

>Command PHONE
>Parameters
USER   'User Name' C D=' '
+
DIV    'Division' C D=' '
>Action KTACTB
```

```
>Command ADDRESS
>Parameters
USER   'User Name' C D=' '
+
DIV    'Division' C D=' '
>Action KTACTB

>Command BEEP
>Parameters
USER   'User Name' C D=' '
+
DIV    'Division' C D=' '
>Action KTACTB

>Command GROUP
>Parameters
USER   'User Name' C D=' '
+
DIV    'Division' C D=' '
>Action KTACTB

>Name KTDEFMB

>Browse Who_CERN 'CERN Members Information' KTSOBJ ktsbro%c
 List

>Class /Division 'CERN Division' big_div sm_div
 List

>Class /Group 'CERN Group' big_gr sm_gr
 List

>Class Usr User big_usr sm_usr
 Phone       .  'Phone [this] [that]'
 Address     .  'Address [this] [that]'
 Beep        .  'Beep [this] [that]'
 All         .  'Phone [this] [that]';'Address [this] [that]';'Beep [this] [that]'
'/Phone...'  .  '-Phone [this] [that]'
'Address...' .  '-Address [this] [that]'

>Icon_bitmaps

#define big_div_width 30
#define big_div_height 23
static char big_div_bits[] = {
   ...
   0xa9, 0xaa, 0xaa, 0x3a, 0xff, 0xff, 0xff, 0x3f};
```

```
#define big_gr_width 30
#define big_gr_height 23
static char big_gr_bits[] = {
   ...
   0xa9, 0x20, 0x8a, 0x3a, 0xff, 0xff, 0xff, 0x3f};

#define big_usr_width 30
#define big_usr_height 23
static char big_usr_bits[] = {
   ...
   0xfd, 0xff, 0xff, 0x3f, 0xff, 0xff, 0xff, 0x3f};

#define sm_div_width 20
#define sm_div_height 16
static char sm_div_bits[] = {
   ...
   0x99, 0xb2, 0x0e, 0x91, 0x32, 0x0e, 0x19, 0x73, 0x0f, 0xff, 0xff, 0x0f};

#define sm_gr_width 20
#define sm_gr_height 16
static char sm_gr_bits[] = {
   ...
   0xa9, 0xc0, 0x0e, 0x91, 0x35, 0x0d, 0x69, 0xb2, 0x0e, 0xff, 0xff, 0x0f};

#define sm_usr_width 20
#define sm_usr_height 16
static char sm_usr_bits[] = {
   ...
   0xa9, 0xaa, 0x0e, 0xd1, 0x7f, 0x0d, 0xfd, 0xff, 0x0f, 0xff, 0xff, 0x0f};
```

N.B. In the action menu definition for the class "Usr" we are using construct of the form "`... [this]`" and "`... [that]`". At command execution "[this]" is replaced by the object name, and "[that]" by the "short description text": both have to be returned by the "scan-objects" routine. In our example the object name is the user name and the "short description" is the group.

Figure 3.11 back on page 138 shows what do you get with KUIP/Motif for this example.

To update the browser **"Object window"** for the new browsable "Who_CERN" defined in the CDF, we have to provide the code of the "scan-objects" routine. In our example the requested information (concerning the "next division", "next group" or "next user") is taken either from data statements (concerning the divisions) or from data files (concerning groups and users). We could have used the same data base as for the well-known commands "Phone" and "WHO" available on most systems at CERN, but this might have make the code of the "scan-objects" routine (KTSOBJ) less easy to understand.

The Fortran code of `KTSOBJ` follows:

Example 3 - "scan-objects" routine (KTSOBJ) - "ktsobj.f"

```
*******************************************************************
*                                                                 *
*   Example of a ''scan-objects'' routine.                        *
*                                                                 *
```

```
*   This routine is called by the Kuip Browser for Who_CERN          *
*   to return next object.                                           *
*                                                                    *
*   Division names are data statements (DIV and LDIV)                *
*   Group names are read from files 'CERN_'div'_'gr'.DAT'            *
*   User names are read from files 'CERN_'div'.DAT                   *
*   Files have a predefined and fixed format.                        *
*                                                                    *
**********************************************************************
      SUBROUTINE KTSOBJ(BRNAME,BRCLAS,BRPATH,OBNAME,OBCLAS,STEXT,LTEXT)
      CHARACTER*(*) BRNAME,BRCLAS,BRPATH,OBNAME,OBCLAS,STEXT,LTEXT
*
      CHARACTER*80 FILENAME
      CHARACTER*80 S1, S2, S3
      CHARACTER*20 UNAME
      CHARACTER*16 FNAME
      CHARACTER*36 USER(500)
      CHARACTER*4  GR
      CHARACTER*80 GROUP(50)
      CHARACTER*8  IDENT, ID
*
      PARAMETER (NDIV = 14)
      CHARACTER*4 DIV(NDIV)
      CHARACTER*80 LDIV(NDIV)
*
      DATA DIV    / 'TH', 'PPE', 'ECP', 'CN', 'AT', 'MT'
     +,              'PS', 'SL', 'ST', 'FI', 'PE', 'TIS'
     +,             'DG', 'AS'/
      DATA LDIV   / 'Theoretical Physics'
     +,             'Particle Physics Experiments'
     +,             'Electronics and Computing for Physics'
     +,             'Computing and Networks'
     +,             'Accelerator Technology'
     +,             'Mechanical Technology'
     +,             'Proton Synchrotron', 'SPS + LEP'
     +,             'Technical Support', 'Finance', 'Personnel'
     +,             'Technical Instection & Safety Commission'
     +,             'Directorate-General'
     +,             'Administrative Support'/
*
      DATA LUN     / 93 /
*
      SAVE ICOUNT

*-------------------------------------------------
*     PRINT *, '============> '
*
*     Get object class according to the browser path :
*     ' ' or /              --> division
*     /CN, /ECP, ...        --> group
*     /CN/AS, /CN/CO, ...   --> users
*
```

```
      OBCLAS='Division'
      IS1=INDEX(BRPATH,'/')
      IF (IS1.GT.0) THEN
          IS2=INDEX(BRPATH(IS1+1:),'/')
          IF (IS2.GT.0) THEN
              OBCLAS='Usr'
          ELSE
              OBCLAS='Group'
          ENDIF
          IS2 = IS2 + IS1
      ENDIF
*
      IF(OBNAME.EQ.' ') ICOUNT = 0
*
      IF(OBCLAS.EQ.'Division') THEN
         STEXT='Division'
         IF (ICOUNT.LT.NDIV) THEN
            OBNAME=DIV(ICOUNT+1)
            LTEXT=LDIV(ICOUNT+1)
         ELSE
            OBNAME=' '
         ENDIF
*
      ELSE IF(OBCLAS.EQ.'Group') THEN
         IF (ICOUNT.EQ.0) THEN
*            Open file CERN_'div'_'gr'.DAT
             FILENAME = 'CERN_'//BRPATH(IS1+1:)//'_GR.DAT'
             OPEN(UNIT=LUN, FILE=FILENAME, STATUS='OLD',
     +            IOSTAT=ISTAT)
             IF (ISTAT.NE.0) THEN
                 PRINT *, '*** Cannot open file ',FILENAME
                 OBNAME=' '
                 GOTO 99
             ENDIF
             NGR = 0
   10        READ(LUN,'(A4,3X,A)',END=30) GR, S1
             NGR = NGR + 1
             WRITE (GROUP(NGR),'(A4,3X,A)') GR, S1(1:LENOCC(S1))
             GOTO 10
   30        CONTINUE
             CLOSE (LUN)
         ENDIF

         IF (ICOUNT.LT.NGR) THEN
            IL=LENOCC(GROUP(ICOUNT+1))
            OBNAME=GROUP(ICOUNT+1)(1:4)
            LTEXT=GROUP(ICOUNT+1)(7:IL)
            STEXT='/'//BRPATH(IS1+1:)//' Group'
         ELSE
            OBNAME=' '
         ENDIF
*
```

```
      ELSE IF(OBCLAS.EQ.'Usr') THEN
*        PRINT *, 'Users for division ',BRPATH(IS1+1:IS2-1)
*        PRINT *, '      and group    ',BRPATH(IS2+1:),' :'
         IDENT = BRPATH(IS1+1:IS2-1)
         IDENT(5:) = BRPATH(IS2+1:)
         IF (ICOUNT.EQ.0) THEN
*            Open file CERN_`div'.DAT
             FILENAME = 'CERN_'//BRPATH(IS1+1:IS2-1)//'.DAT'
             OPEN(UNIT=LUN, FILE=FILENAME, STATUS='OLD',
     +            IOSTAT=ISTAT)
             IF (ISTAT.NE.0) THEN
                 PRINT *, '*** Cannot open file ',FILENAME
                 OBNAME=' '
                 GOTO 99
             ENDIF
             NUSR = 0
   50        READ(LUN,10100,END=60) UNAME, FNAME, S2, ID, S3
             IF (ID.EQ.IDENT) THEN
                 NUSR = NUSR+1
                 WRITE (USER(NUSR),'(A20,A16)') UNAME, FNAME
             ENDIF
             GOTO 50
   60        CONTINUE
             CLOSE (LUN)
         ENDIF
         IF (ICOUNT.LT.NUSR) THEN 138

            UNAME = USER(ICOUNT+1)(1:20)
            FNAME = USER(ICOUNT+1)(20:36)
            OBNAME=UNAME
            STEXT=BRPATH(IS1+1:IS2-1)  ! Div. (used for [that] in menus)
            LTEXT=FNAME                ! Firstname
         ELSE
            OBNAME=' '
         ENDIF
*
      ENDIF
*
      ICOUNT = ICOUNT + 1
*
10000 FORMAT('P=',A64,A4,A10)
10100 FORMAT('P=',A20,3X,A16,1X,A20,A8,A10)
*
   99 RETURN
      END
```

This routine could have been written in C with the following skeleton:

Example 3 - “scan-objects” routine skeleton in C

```
char **ktsobj( brobj_name, brcls_name, bpath, n )
     char *brobj_name;  /* browsable name <> BRNAME */
     char *brcls_name;  /* browsable class name <> BRCLAS */
     char *bpath;       /* current directory path <> BRPATH */
     int n;             /* object position (0 the first time) */
{
  static char     *obj_desc[4];

    ...

  return obj_desc;   /* obj_desc[0] --> object name <> OBNAME
                        obj_desc[1] --> class name <> OBCLAS
                        obj_desc[2] --> short text description <> STEXT
                        obj_desc[3] --> long text description <> LTEXT */
}
```

The browsable “Who_CERN” is a “single instance” one (see section 3.3.2). The “scan-browables” routine is optional, and we use it just to fill the entry “Path:” (top of the browser) and “File” (bottom of the browser) with more meaningful values that the ones which are eventually put by default.

Example 3 - “scan-browsables” routine (ktsbro) - ”ktsbro.c”

```
/***********************************************************************
 *                                                                     *
 *   Example of a ‘‘scan-browables’’ routine.                          *
 *                                                                     *
 *   This routine is called by the Kuip Browser for Who_CERN.          *
 *                                                                     *
 ***********************************************************************/

char **ktsbro( class_name, first )
     char *class_name;
     int first;
{
   static char *path_desc[2];
   static char  root[80];

   path_desc[0] = NULL;
   path_desc[1] = NULL;

   if (first) {
      strcpy(root, "root=$CERN file=ČERN divisions, groups and members");
      path_desc[0] = "Who_CERN";
      path_desc[1] = root;
   }
   return path_desc;
}
```

The following is the action routine (KTACTB), coded in Fortran, for the commands defined in the CDF (“Phone”, “Address”, “Beep”, “Group”). We are using the KUIP subroutine “KUMESS” in order to display a message using the standard Motif “MessageDialog” widget (it is just a “print” statement for the terminal version).

Example 3 - Action Routine KTACTB - (“ktactb.f”)

```
      SUBROUTINE KTACTB
*
      CHARACTER*32 CHPATH, CVAL, CDIV
*
      PARAMETER (NDIV = 14)
      CHARACTER*80 FNAME(NDIV), FILE, LINE
*
      DATA LUN/93/
      DATA FNAME/'CERN_TH.DAT', 'CERN_PPE.DAT', 'CERN_ECP.DAT'
     +,          'CERN_CN.DAT', 'CERN_AT.DAT', 'CERN_MT.DAT'
     +,          'CERN_PS.DAT', 'CERN_SL.DAT', 'CERN_ST.DAT'
     +,          'CERN_FI.DAT', 'CERN_PE.DAT', 'CERN_TIS.DAT'
     +,          'CERN_DG.DAT', 'CERN_AS.DAT' /
*
* Retrieve command name and number of parameters
*
      CALL KUPATL (CHPATH,NPAR)
*
* Retrieve parameter values
*
      CDIV = ' '
      CALL KUGETC (CVAL, ILEN)
      IF (NPAR.EQ.2) THEN
          CALL KUGETC (CDIV, ILD)
      ENDIF
*
      IFOUND=-1
      IF (CDIV.NE.' ') THEN
          FILE = 'CERN_'//CDIV(1:ILD)//'.DAT'
          CALL GETVAL (CHPATH, CVAL, FILE, ISTAT)
          IF (ISTAT.EQ.0) IFOUND=0
      ELSE
          DO 10 I=1,NDIV
             CALL GETVAL (CHPATH, CVAL, FNAME(I), ISTAT)
             IF (ISTAT.EQ.0) IFOUND=0
   10 CONTINUE
      ENDIF
*
      IF (IFOUND.EQ.-1) THEN
*         PRINT *,'*** Cannot find user : ', CVAL(1:ILEN)
          CALL KUMESS ('*** Cannot find user : '//CVAL(1:ILEN), 0)
          CALL KUMESS ('    in division: '//CDIV, 2)
      ENDIF
*
  99  RETURN
      END


      SUBROUTINE GETVAL (CHPATH, CVAL, FILENAME, IFOUND)
*
      CHARACTER*(*) CHPATH, CVAL, FILENAME
      INTEGER      IFOUND
```

```
*
      CHARACTER*80 UNAME, FNAME, PHONE1, PHONE2, BEEP
      CHARACTER*80 DIV, GROUP, BAT, OFFICE

      DATA LUN/93/
*
      IFOUND = -1
      IL = LENOCC(FILENAME)
      OPEN(UNIT=LUN,FILE=FILENAME(1:IL),STATUS='OLD',IOSTAT=ISTAT,
     +     ERR=99)
      IF (ISTAT.NE.0) THEN
         GOTO 40
      ENDIF
*
   20 READ(LUN,10000,END=40)
     +     UNAME, FNAME, PHONE1, PHONE2, BEEP, DIV, GROUP, BAT, OFFICE
      ILEN = LENOCC(CVAL)

      IF (UNAME.EQ.CVAL(1:ILEN)) THEN
          IFOUND = 0
          IL = LENOCC(UNAME)
          IL1 = LENOCC(FNAME)
          CALL KUMESS ('===> User '//UNAME(1:IL)//' '//FNAME(1:IL1), 0)
          IF (CHPATH.EQ.'PHONE') THEN
              IL = LENOCC(PHONE1)
              IL1 = LENOCC(PHONE2)
              IF (PHONE1(1:IL).NE.' ') THEN
                  CALL KUMESS (
     +            '     Phone : '//PHONE1(1:IL)//'  '//PHONE2(1:IL1), 2)
              ELSE
                  CALL KUMESS ('     No phone.', 2)
              ENDIF
          ELSE IF (CHPATH.EQ.'ADDRESS') THEN
              IL = LENOCC(BAT)
              IL1 = LENOCC(OFFICE)
              IF (BAT(1:IL).NE.' ') THEN
                  CALL KUMESS (
     +            '     Bat. '//BAT(1:IL)//', off. '//OFFICE(1:IL1), 2)
              ELSE
                  CALL KUMESS ('     No address at CERN.', 2)
              ENDIF
          ELSE IF (CHPATH.EQ.'BEEP') THEN
              IL = LENOCC(BEEP)
              IF (BEEP(1:IL).NE.' ') THEN
                  CALL KUMESS ('     Beep : '//BEEP(1:IL), 2)
              ELSE
                  CALL KUMESS ('     No beep.', 2)
              ENDIF
          ELSE IF (CHPATH.EQ.'GROUP') THEN
              IL = LENOCC(DIV)
              IL1 = LENOCC(GROUP)
              IF (DIV(1:IL).NE.' ') THEN
                  CALL KUMESS (
```

```
     +            '     Div.: '//DIV(1:IL)//', group: '//GROUP(1:IL1), 2)
               ELSE
                   CALL KUMESS ('     No group at CERN.', 2)
               ENDIF
           ENDIF
           GO TO 40
       ELSE
           GO TO 20
       ENDIF
*
   40 CONTINUE
      CLOSE (LUN)
*
10000 FORMAT('P=',A20,3X,A16,1X,A4,3X,A4,1X,A8,A4,A4,A4,1X,A5)
*
 99   RETURN
      END
```

The following is the application main program in Fortran:

Example 3 - Main Program - "ktestb_main.f"

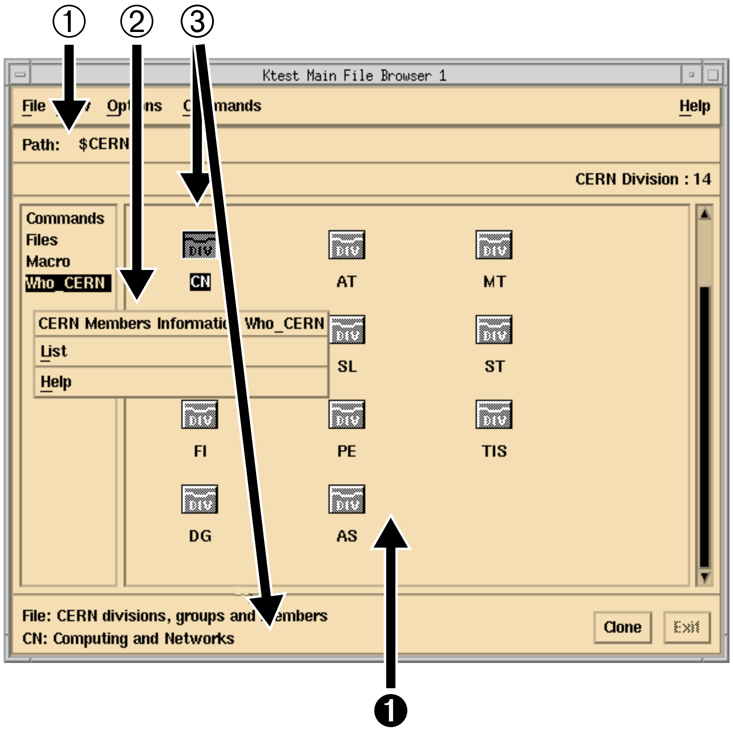
```
      PROGRAM KTEST
*
* Basic KUIP Application with MOTIF / NOT linked to HIGZ
*
      COMMON/PAWC/PAW(500000)
      CALL INITC
*
* Initialize PAW
*
      CALL MZEBRA(-3)
      CALL MZPAW(500000,' ')
*
* Initialize KUIP with NWORDS words as minimum division size
*
      NWORDS=20000
      CALL KUINIT(NWORDS)
*
* Create user command structure from definition file (CDF)
*
      CALL VECDEF
      CALL KTDEF          ! Command definition part
*                         ! (code generated from the CDF compilation)
*
* Gives access to KUIP browsers for commands, files and macros
*
      CALL KUIDFM
*
```

```
* Gives access to the Who_CERN browser (from definition file)
*
      CALL KTDEFMB        ! Browsable and class definitions
*                         ! (code generated from the CDF compilation)
*
* Execute some KUIP Initialization Commands
*
      CALL KUEXEC('PROMPT ''KTEST >''')
      CALL KUEXEC('LAST 0')
*
* Give control to MOTIF
*
      CALL KUWHAM ('Ktestb')
*
      END
```

The class-name of the application (for X Resources) has been set to "Ktestb". The link command for `ktestb` is:

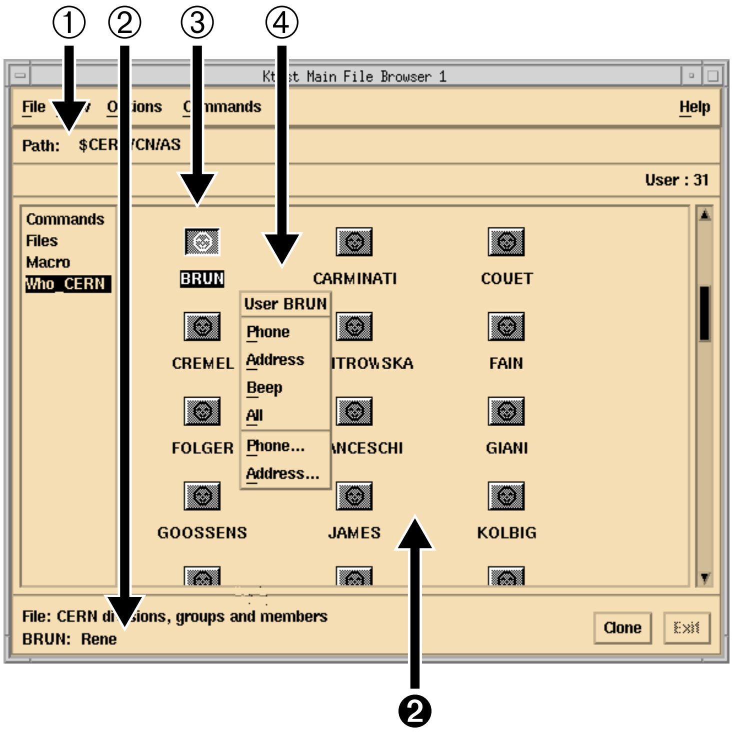
```
f77 -o ktestb ktestb_main.f ktcdfb.c ktactb.f ktsobj.f ktsbro.c `cernlib -G Motif`
```

Figure 3.12 and 3.13 show the browsable "Who_CERN" in two different states.



"Who_CERN" has just been selected in the **"Browsable window"**. The entry "Path:" ① is filled with "$CERN" (top directory). The pop-up menu for this browsable ② is displayed (with the 2 items "List" and "Help"). In the **"Object window"** ❶ the list of all CERN divisions is displayed. One division (CN) is selected and its "long text" description ("CN: Computing & Networks") is written at the bottom of the browser ③.

Figure 3.12: Who_CERN Browsable - First State (Example 3)



First CN division and then group AS have been selected in the **"Object window"** ❷ with a <double click>. The list of all members of group AS in division CN is displayed. One user ("BRUN") ③ has been selected and the pop-up menu for the class "Usr" ④ (with the 6 items: "Phone", "Address", "Beep", "All", "Phone..." and "Address...") is displayed. The entry "Path:" ① is filled with "$CERN/CN/AS" and the "long text" description for user BRUN ("BRUN: Rene") ② is written at the bottom of the browser.

Figure 3.13: Who_CERN Browsable - Second State (Example 3)

4 KUIP calling sequences

In the specification of the KUIP calling sequences we use the following conventions:

- Names starting with `CH` are type `CHARACTER*(*)`.
- Names starting with `DP` are type `DOUBLE PRECISION`.
- Names starting with `LG` are type `LOGICAL`.
- Names starting with `XT` are type `EXTERNAL`.
- Other names starting with `I-N` are type `INTEGER`.
- Any other names are type `REAL`.

- Input/Output parameters are indicated by a "*" preceding and following the name, for example `*IVAL*`.
- Output-only parameters are indicated by a "*" following the name, for example `ISTAT*`.
- Otherwise parameters are input-only.

Prototypes for the C functions are available in the header file `/cern/pro/include/kuip.h`.

4.1 Framework

4.1.1 Initialization

KUIP requires a varying number of initialization calls depending whether the application uses HIGZ graphics and whether a command line or Motif interface is requested. The calls are documented in the order in which they have to be executed. See section 4.5 for examples which calls are mandatory for which mode.

`CALL` **INITC**

Action: Initialize Fortran-C interface.

`INITC` is not specific to KUIP but rather needed for every program mixing Fortran and C. If `INITC` is not called as the first statement in the Fortran main program, it will crash on VM/CMS and MVS sooner or later with an obscure error message.

`CALL` **KUINIT** `(NWORDS)`

Parameter Description:

`NWORDS` Number of words to be allocated as minimum size of KUIP division in the ZEBRA store.

Action: Initialize KUIP kernel.

Remark:

`MZEBRA` and `MZPAW` must have been called before `KUINIT` (see examples in section 4.5).

`CALL` **VECDEF**

Action: Define command structure for `VECTOR` menu.

Remarks:

`VECDEF` should be called before any application specific command definition routines in order to keep the built-in command menus `KUIP`, `MACRO`, and `VECTOR` together as the first entries in the online help.

The call to `VECDEF` can be omitted if the application has no use for vectors.

`CALL` **KUIDFM**

Action: Define KUIP/Motif built-in browsables `Commands`, `Files`, and `Macro`.

Remarks:

`KUIDFM` should be called before any application specific browser definition routines in order to keep the built-in browsables as the first entries in the **"Browsable window"**.

`KUIDFM` should only be called for KUIP/Motif applications.

`CALL` **KUTERM** `(XTTERM)`

`CALL` **KUGRFL** `(XTTERM)`

Parameter Description:

`XTTERM` Routine without parameters.

Action: Define routines to be called before writing to the output stream (`KUTERM`) or to flush the graphics output buffer (`KUGRFL`).

Depending on the device the `XTTERM` routine has to switch from graphics to text mode (for graphics terminals) or to flush the graphics output buffer (for e.g. X11). Applications using HIGZ graphics have to call `KUTERM(IGTERM)`.

`KUGRFL` defines the routine which is called after the execution of each command in order to flush the graphics output buffer. Applications combining HIGZ graphics with KUIP/Motif have to call `KUGRFL(IGTERM)`.

```
CALL KUINIM (CHCLASS)
```

Parameter Description:

CHCLASS	Application class name used for looking up application resources (see section 2.8.4). If CHCLASS is blank the class name defaults to Mkuip.

Action: Initialize KUIP/Motif and create the **"Main Browser"** and **"Executive Window"**.

An explicit call to KUINIM is only necessary if the application needs to gain control between the Motif initialization and entering the KUWHAM loop. Otherwise the Motif initialization will be done by KUWHAM. Under these circumstances KUINIM has to be called explicitly:

- The application uses graphics: HIGZ can only be initialized after KUIP/Motif. See the example on page 192 for the initialization steps when using HIGZ.
- The application prints a welcome message: Any terminal output before the call to KUINIM will go to the shell window from which the application program was started instead of the **"Executive Window"**.
- The application executes a logon macro: Only after calling KUINIM the $STYLE function (see page 34) will return the value XM.

4.1.2 Command execution

```
ISTAT = KUEXEL (CHLINE)
```

```
ISTAT = KUEXEC (CHLINE)
```

```
int status = ku_exel (const char* line)
```

```
int status = ku_exec (const char* line)
```

Parameter Description:

CHLINE	Command line to be executed.
ISTAT	Status code, 0 if successful.

Action: Execute a command line following the syntax rules as described in section 2.2. The status code is also stored in IQUEST(1).

KUEXEC and KUEXEL can also be called from inside an action routine. The difference between the two functions is that in Motif mode KUEXEC echoes the command line in the **Transcript Pad** of the **"Executive Window"**.

Examples:

```
CALL KUEXEL('SET/PROMPT ''PAW >''')
CALL KUEXEL('EXEC PAWLOGON')
```

```
CALL KUWHAT
```

```
CALL KUWHAG
```

Action: Give control to the KUIP command loop. The user is prompted for new commands until one of the commands `KUIP/EXIT` or `KUIP/QUIT` is executed.

When calling `KUWHAT` instead `KUWHAG` the styles `G` and `GP` are not provided and the application does not need to load any HIGZ code.

```
CALL KUWHAM (CHCLASS)
```

Parameter Description:

`CHCLASS` Application class name used for looking up application resources (see section 2.8.4). If `CHCLASS` is blank the class name defaults to `Mkuip`.

Action: Call `KUINIM(CHCLASS)` if it hasn't been called explicitly and give control to the Motif event loop.

```
void ku_last (const char* line)
```

Parameter Description:

`line` Command line.

Action: Append `line` to the `last.kumac` file.

4.1.3 Customization

There are a few more initialization calls which allow customization of KUIP. They should be called in the position of `KUTERM`.

```
CALL KUEXIT (XTEXIT)
```

```
CALL KUQUIT (XTQUIT)
```

Parameter Description:

`XTEXIT` User exit routine.
`XTQUIT` User quit routine.

Action: Define parameterless user routines called by the `EXIT` and `QUIT` commands, respectively, before breaking the command input loop.

`CALL` **KUBREK** `(XTBREK)`

Parameter Description:

`XTBREK` User break routine.

Action: Define a user routine called after intercepting a keyboard interrupt and before returning to the command input prompt.

`CALL` **KUSER** `(XTUSER)`

Parameter Description:

`XTUSER` User command routine.

Action: Define a "user command" routine which allows to modify the command line input before execution.

This routine is called for each command line typed by the user. The input line can be retrieved from the common block `/KCPARC/`. If the command should not be executed at all `IQUEST(1)` can be set to a non-zero value. Otherwise the command line can be modified and stored back into `/KCPARC/`.

Example of a user command routine

```
      SUBROUTINE XTUSER
*
      COMMON /KCPARC/ CMDLIN
      CHARACTER CMDLIN*255
*
      COMMON /QUEST/ IQUEST(100)
*
      CHARACTER CHTEMP*255
*
      IF(CMDLIN(1:1).GE.'1'.AND.CMDLIN(1:1).LE.'9') THEN
*
*  Each command starting with a non-zero number (e.g. 5) is treated  as a
*  VECTOR/PRINT of a vector corresponding to that number (e.g. VECTOR/PRINT V5).
*
        CHTEMP='VECTOR/PRINT V'//CMDLIN
        CMDLIN=CHTEMP
      ELSEIF(CMDLIN(1:1).EQ.'0') THEN
*
*  If number starts with zero (0), skip the execution of the command.
*
        IQUEST(1)=1
      ENDIF
*
      END
```

Remark:

The user command routine is active only in style `C`.

`CALL` **KUSIGM** `(XTSIGM)`

Parameter Description:

`XTSIGM` SIGMA entry routine.

Action: Define the SIGMA entry routine. In order to enable the `$SIGMA` function (see page 38) the following initialization code must be present:

```
EXTERNAL SIGMAE
.....
CALL SIGINI1
CALL KUSIGM(SIGMAE)
```

`SIGINI1` and `SIGMAE` require linking to the PAWLIB library.

`CALL` **KUMLOC** `(XTMLOC)`

Parameter Description:

`XTMLOC` KUIP/Motif locator call-back routine.

Action: Define the locator call-back routine for KUIP/Motif. If a graphics application wants to handle mouse operations in HIGZ windows it can register a routine which is called for each related X-window event. The form of the call-back routine is

```
SUBROUTINE XTMLOC(IX,IY,IEVENT,IWKID)
```

`IX,IY`	Pointer coordinates in pixels relative to the graphics window.	
`IEVENT`	Event type:	
	`+999,-999`	Enter or Leave window event.
	`0`	Pointer motion event.
	`+1,-1`	Left mouse button Press or Release event.
	`+2,-2`	Middle mouse button Press or Release event.
`IWKID`	HIGZ window identifier.	

Remarks:

Dragging, i.e. pointer movements while a mouse button is pressed, has to be recognized by the routine itself. The right mouse button is reserved for pop-up menus and will not trigger calls to `XTMLOC`.

`CALL` **KULUN** `(LUNO)`

Parameter Description:

`LUNO` Base of logical units reserved for KUIP.

Action: KUIP requires 5 consecutive Fortran logical unit numbers for internal use. By default the units 11–15 are reserved. `KULUN` allows to shift the base unit number to a value different from `LUNO=11`.

`CALL KUFDEF (CHSYNP,CHGUID,CHFACE,CHCALL,XTFUNC)`

Parameter Description:

CHSYNP	Synopsis of function usage
CHSYNP,CHGUID	One line of explanation text shown by HELP FUNCTIONS together with synopsis
CHFACE	Description of the application function's interface
CHFACE	Description of the external function's calling sequence
XTFUNC	External function

Action: Define an application specific function.

KUFDEF allows to add application specific functions which then can be used just like any KUIP defined system function (see section 2.4).

The function interface is described in CHFACE in the format

T=*name*(*T*,...)

where *T* is a single letter variable name with the implicit typing scheme

i,j,k	INTEGER
l,m,n	LOGICAL
x,y,z	REAL
d,e,f	DOUBLE PRECISION
s,t,u	CHARACTER

For example, 'l=$HEXIST(i)' says that the function name is $HEXIST and that it takes one integer argument and returns a boolean value as 0 or 1.

CHCALL has the same form as CHFACE and describes the calling sequence of the external XTFUNC. The definition of $HEXIST is then achieved by

```
      CALL KUFDEF('$HEXIST(id)',
     +     '1 if histogram ID exists or 0 otherwise',
     +     'l=$HEXIST(i)',
     +     'l=HEXIST(i)',HEXIST)
```

The KUIP expression parser knows only one numeric data type and does not distinguish between integer or floating point value. The correct typing is, however, necessary in CHCALL and CHFACE must refer to the matching variables.

There is no need to have the one-to-one correspondance between CHFACE and CHCALL as for HEXIST. For example,

```
      CALL KUFDEF('$HINFO(id,''ENTRIES'')',
     +     'Number of entries',
     +     'k=$HINFO(i,S=ENTRIES)',
     +     '-=HNOENT(i,k)',HNOENT)
```

calls HNOENT(10,k) and returns the value of k. The "-" in CHCALL indicates that XTFUNC is a SUBROUTINE.

The construct S=*string* in CHFACE allows to define different external functions with different calling sequences using the same interface name. While

```
[n] = $HINFO([id],'ENTRIES')
```

calls HNOENT,

```
[s] = $HINFO([id],'SUM')
```

can call a different routine by defining

```
      CALL KUFDEF('$HINFO(id,''SUM'')',
     +     'Total histogram content',
     +     'x=$HINFO(i,S=SUM)',
     +     '-=HSUM(x,$,i)',PHINFO)
```

The name in CHCALL has no real significance and is free to be used for specifying a CHARACTER constant which should be passed at the position of the "$", i.e. call PHINFO(x,'HSUM',i).

The function can also access array elements and return their value. For example in

```
      CALL KUFDEF('$GRAFINFO(''WNXMIN'')',
     +     'Lower X limit of window in current NT',
     +     'x=$GRAFINFO(S=WNXMIN)',
     +     '-=NTWN(I,$,x0)',IGQWK)
```

the routine IGQWK fills an array passed as third argument, and x0 tells that $GRAFINFO('WNXMIN') should return the first array element. The indices run from 0 to 9, i.e. x1 refers to the second array element. The routine must not overwrite more than 10 elements.

The "I" indicates that the call to IGQWK requires a dummy INTEGER as first argument. CHCALL would also allows to pass an integer constants by defining e.g. '-=NTWN(I=17,$,x0)'. Likewise LOGICAL, REAL, and CHARACTER constants can be specified by L=0 for .FALSE., L=1 for .TRUE., R=*number*, or C=*string*, respectively.

The character variable u is special in the respect that if it is used in CHFACE as argument, the content is converted to uppercase before it is passed to XTFUNC. For example, with

```
      CALL KUFDEF('$GRAFINFO(''?attr'')',
     +     'HPLOT/HIGZ attribute (see HELP SET for valid names)',
     +     'x=$GRAFINFO(u)',
     +     '-=HPLSET(u,x)',HPLSET)
```

the user can type $grafinfo('?hcol') but the call will be set up as HPLSET('?HCOL',x).

Despite the apparent flexibility of the CHFACE and CHCALL descriptions there are some restrictions. The interface classes, i.e. the argument and return value types have to be built into the KUIP expression parser. At the moment for CHFACE the following combinations are implemented where S stands for a string and X stands for a numeric argument:

- S = $fun()
- S = $fun(S)
- S = $fun(X)
- S = $fun(X,S)
- X = $fun(S)
- X = $fun(X)
- X = $fun(X,S)

A similar restriction applies to the call classes. For CHCALL the following combinations are provided:

- fun(X)
- fun(X,X)

- fun(S,X)
- fun(S,S)
- fun(S,S,X,X)
- fun(S,S,X,X,X)
- fun(X,S,X)
- fun(X,S,S)
- fun(X,S,X,X,X,X,X,X,X,X)

More interface and call classes can easily be added on request. XTFUNC must be either a SUBROUTINE, or a INTEGER or LOGICAL function. In order to call a REAL FUNCTION a wrapper routine (such as PHINFO) must be used. A wrapper routine can also be employed if the call class of the target routine is not implemented.

4.1.4 Command definition

In Fortran output mode KUIPC generates calls to the routines described in this section. These calls construct the command tree in memory. Users usually do not need to understand except for very special applications or if they suspect a bug in the CDF translation.

CALL **KUCMD** (CHPATH,CHITEM,CHOPT)

Parameter Description:

CHPATH	Command or menu path (absolute or relative)
CHITEM	Name for new item
CHOPT	Operation selector:
	'C' Create a new command or menu with name CHITEM
	'SW' Switch to menu CHPATH
	'D' Delete command CHPATH

Action: KUCMD manipulates commands and menus.

Remark:

The semantics of KUCMD is weird: when creating a new item there is no way to tell whether it should be a command or a menu. Therefore the item is first treated as a command. Only when there is a KUCMD(...,'SW') later-on the command is converted into a menu.

CALL **KUPAR** (CHPATH,CHNAME,CHPROMPT,CHTYPE,CHOPT)

Parameter Description:

CHPATH	Command path
CHITEM	Parameter name
CHPROMPT	Prompt string
CHTYPE	First character parameter type, remaining characters parameter attributes
CHOPT	Operation selector must be 'S'

Action: Define a new parameter for a command.

`CALL` **KUPVAL** `(CHPATH,CHNAME,IVAL,RVAL,CHVAL,CHOPT)`

Parameter Description:

`CHPATH`	Command path
`CHITEM`	Parameter name
`IVAL`	Integer value
`RVAL`	Real value
`CHVAL`	Character value
`CHOPT`	Operation selector:
	`'D'` Default value
	`'L'` Lower range value
	`'U'` Upper range value
	`'V'` Value list
	`'-x'` Option *x*

Action: Set parameter values.
The parameter type defined by `KUPAR` decides whether `IVAL`, `RVAL`, or `CHVAL` is relevant.

`CALL` **KUGUID** `(CHPATH,CHGUID,NLINES,CHOPT)`

Parameter Description:

`CHPATH`	Command path
`CHGUID`	Guidance text with definition
`NLINES`	Number of valid lines in `CHGUID`
`CHOPT`	Operation selector:
	`'S'` Store: copy from `CHGUID` into command structure
	`'Q'` Query: copy from internal structure into `CHGUID`

Remark:
There is a common block

```
CHARACTER*80 CHGUID
COMMON /KCGUID/ CHGUID(199)
```

which can be used as temporary space for the guidance text.

`CALL` **KUACT** `(CHPATH,XTACTN)`

`CALL` **KUACH** `(CHPATH,XTHELP)`

Parameter Description:

`CHPATH`	Command path
`XTACTN`	Action routine
`XTHELP`	User help routine

Action: Define the action routine (`KUACT`) or user help routine (`KUACH`) connected to the command name.

4.1.5 Environment

```
CALL KUQENV (*CHLINE*)
```

```
char** env = ku_qenv ()
```

Parameter Description:

`CHLINE`	Command line
`env`	NULL-terminated array of command lines

Action: Inquire current setting of KUIP environment commands `HOST_SHELL`, `HOST_EDITOR` etc.

`KUQENV` returns the commands which will be necessary to restore the current environment in the next session. `CHLINE` has to be initialize to blank in order to return the first command line. If `CHLINE` is returned as blank the last command line has been passed. For example, these commands can be written into a macro file which is then executed the next time the application starts:

```
      CHARACTER CHLINE*80
      CALL KUOPEN(LUN,'restore.kumac','UNKNOWN',ISTAT)
      CHLINE=' '
 1    CONTINUE
      CALL KUQENV(CHLINE)
      IF(CHLINE.NE.' ') THEN
        CALL KUWRIT(LUN,CHLINE)
        GOTO 1
      ENDIF
      CALL KUCLOS(LUN,' ')
```

The equivalent code in C would be:

```
FILE *stream = fopen( "restore.kumac", "w" );
char** env = ku_qenv();
int n;

for( n = 0; env[n] != NULL; n++ ) {
  fprintf( stream, "%s\n", env[n] );
}

fclose( stream );
```

```
LGQEXE = KUQEXE (CHNAME)
```

```
char* path = ku_qexe (const char* name)
```

Parameter Description:

`CHNAME`	Executable name

Action: Inquire about an executable module.

`Unix`	: Search for file `name` in all directories listed in `PATH` environment variable.
`MS-DOS`	: Search for file `name.EXE` in all directories listed in `PATH` environment variable.
`VMS`	: Search for file `name.EXE` in all directories listed in `KUIPPATH` logical name.
`VM/CMS`	: Search for file `name MODULE` on all accessed disks.

`KUQEXE` tests whether the executable exists. `ku_qexe` returns the full path name of the executable or `NULL` if it does not exist.

`CALL` **KUQVAR** `(CHEXPR,CHVAL*)`

`char* value =` **ku_expr** `(const char* expr)`

`char* value =` **ku_eval** `(const char* expr)`

`int status =` **ku_math** `(const char* expr, double* value)`

`int value =` **ku_bool** `(const char* expr)`

Parameter Description:

`CHEXPR`	Expression to be evaluated
`CHVAL`	Evaluation result

Action: Evaluate an expression and return the result.

`KUQVAR` and `ku_expr` can be used to inquire alias expansions, macro variables, etc. They yield the same result as in a macro variable assignment. If the input does not qualify as a proper expression, evaluation continues as a garbage expression (see section 2.6.4)

`ku_eval` requires a proper (arithmetic or string) expression or returns `NULL` otherwise. The pointers returned by `ku_expr` and `ku_eval` are allocated.

`ku_math` requires an arithmetic expression or returns a non-zero `status` value otherwise.

`ku_math` requires a boolean expression and returns 0 for false, 1 for true, or -1 for invalid expressions.

`CALL` **KUVAR** `(CHLINE)`

Parameter Description:

`CHLINE`	Macro variable assignment *var*=*value*

Action: Assign value to a macro variable.

`int status =` **ku_qmac** `(const char* mname)`

Parameter Description:

`mname`	Macro name	
`status`	`-1`	The currently executing macro was invoked from the command line.
	`1`	There is a macro *mname* in the same `.kumac` file.
	`0`	*mname* must be the name of a `.kumac` file.

Action: Test status of macro name.

`ku_qmac` is for the special case that an application wants to overload the `EXEC` command. For example, `EXEC` is set up to invoke the command `CMZ/EXEC` which first checks whether *mname* is the name of a deck. If that is the case, the deck is extracted into a file before calling `MACRO/EXEC`.

On the other hand "`EXEC` *mname*" has the semantics that it looks first for a macro *mname* defined in the same file. Therefore `CMZ/EXEC` has to test `ku_qmac`, and if it returns 1 `MACRO/EXEC` *mname* has to be called without looking for a deck of the same name.

`CALL` **KUSERID** `KUSERID(CHNAME*)`

`char* name =` **k_userid** `()`

Parameter Description:

`CHNAME`	User name

Action: Inquire the user account name.

`NCMD =` **KUSTAT** `(IWHICH)`

Parameter Description:

`IWHICH` Selection switch:

`0`	number of commands in total (macro + keyboard)
`1`	number of different commands in total
`2`	number of commands from keyboard
`3`	number of different commands from keyboard
`4`	number of commands inside macro
`5`	number of different commands inside macro

otherwise reset counters

Action: Inquire statistics about command usage.

Remark:

`KUSTAT(-1)` should be called after `KUINIT` because there a number of `KUEXEL` calls are used for initialization.

4.2 Action routines

4.2.1 Argument retrieval

The action routine specified in the CDF for each command is called without any arguments. The following routines have to be used in order to retrieve the command line arguments.

Note that KUIP does not enforce a correspondence between the parameter description in the CDF and the actual argument retrieval. For example, an argument declared as character string can be retrieved as integer. In fact, for the command line interface the parameters do not need to be described at all! However, this is not recommended practice because from the CDF description derives both the online help and the "Command Argument Panel" for the Motif interface.

```
NPAR = KUNPAR ()
```

```
int npar = ku_npar ()
```

Action: Return the number of arguments supplied by the user on the command line, i.e. not counting parameters for which the KUGETx routines will return the default value.

Remark:

If the use of named arguments (see section 2.2.2.1) creates "holes" in the value list actually specified by the user, KUNPAR returns the position of the last user-provided argument. For example, if the command is defined as

```
CMD a b [ c d e f ]
```

and the user enters "CMD 1 2 e=17" then KUNPAR returns "5".

```
CALL KUGETI (*IVAL*)
```

```
CALL KUGETR (*RVAL*)
```

```
int value = ku_geti ()
```

```
double value = ku_getr ()
```

Parameter Description:

IVAL	Integer argument value.
RVAL	Real argument value.

Action: KUGETI and KUGETR retrieve the next command argument as INTEGER or REAL value, respectively. The value returned is

- either the value provided by the user, possibly the result of an arithmetic expression evaluation (see section 2.6.1),
- or the default value for optional parameters,
- or the initial content of IVAL and RVAL if the parameter is optional and does not have a default.

Optional parameters without default value are not supported by the C functions.

```
CALL KUGETC (*CHVAL*,LENGTH*)
```

```
CALL KUGETS (*CHVAL*,LENGTH*)
```

```
char* value = ku_getc ()
```

```
char* value = ku_gets ()
```

Parameter Description:

CHVAL	String argument value.
LENGTH	Logical length of CHVAL, i.e. ignoring trailing blanks.

Action: KUGETC retrieves the next command argument as CHARACTER string value converted to uppercase while KUGETS does not apply any case conversion.

Remark:

The character pointers returned by ku_getx() are static and are guaranteed to be valid only up to the next command execution.

```
CALL KUGETF (*CHVAL*,LENGTH*)
```

```
char* value = ku_getf ()
```

Parameter Description:

Same as for KUGETC.

Action: Retrieve next command argument as filename. For KUGETF the case conversion applied depends on the operating system:

- If filenames are case-insensitive, the string is converted to uppercase.
- If filenames are case-sensitive (Unix),
  - the string is converted to lowercase, if FILECASE -CONVERT is active,
  - or the string is left as it is, if FILECASE -KEEP is active.

Remark:

For KUGETC, KUGETS, and KUGETF applies the same order as for KUGETI and KUGETR to determine the value to be returned.

```
CALL KUGETL (CHVAL*,LENGTH*)
```

```
char* value = ku_getl ()
```

Parameter Description:

Same as for KUGETC.

Action: Retrieve the next item from an argument list. KUGETL returns the next item from a comma separated argument list. KUGETL must be called after KUGETC, KUGETS, or KUGETF in order to determine the case conversion. The value LENGTH=0 is returned when the list is exhausted.

Example for using KUGETL

```
      CALL KUGETC(CHITEM,L)
 1    CONTINUE
      CALL KUGETL(CHITEM,L)
      IF(L.NE.0) THEN
         ...
         GOTO 1
      ENDIF
```

Remark:

The CHARACTER variable used in the KUGETC call does not need to fit the complete argument list and it can be reused immediately afterwards.

```
CALL KUGETE (CHVAL*,LENGTH*)
```

```
char* value = ku_gete ()
```

Parameter Description:

Same as for `KUGETC`.

Action: Retrieve the remaining command line arguments separated by single blanks.

```
CALL KUGETV (CHVECT*,LLO*,LHI*)
```

Parameter Description:

See `KUVECT` on page 176.

Action: Retrieve next command argument as vector specification.

`KUGETV` is a combination of calling `KUGETC`, `KUVECT`, and printing an error message if the vector does not exist.

```
CALL KUSPY (CHOPT)
```

```
void ku_spy (const char* option)
```

Parameter Description:

`CHOPT`	Option `'ON'` or `'OFF'`.

Action: Turn argument spying mode on or off.

`KUSPY` allows to mark an argument position and to return to it later-on after "spying" on successive arguments.

Example:

Suppose that a command has five parameters but only the third one specifies whether the second argument is an integer or a string value. Therefore the third argument must be examined before the final decision of using `KUGETI` or `KUGETC` to retrieve the second argument.

Example for using KUSPY

```
      SUBROUTINE USESPY
      CHARACTER*8 CHVAL1,CHVAL2,CHVAL3,CHVAL4,CHVAL5
      ...
      CALL KUGETC(CHVAL1,NCH1)
*--- set mark that we might have to reexamine arg2
      CALL KUSPY('ON')
      CALL KUGETC(CHVAL2,NCH2)
      CALL KUGETC(CHVAL3,NCH3)
      IF(CHVAL3.EQ.'-I') THEN
         CALL KUSPY('OFF')
*--- get arg2 again, this time as integer
         CALL KUGETI(IVAL2)
         CALL KUGETC(CHVAL3,NCH3)
      ENDIF
```

```
CALL KUGETC(CHVAL4,NCH4)
CALL KUGETC(CHVAL5,NCH5)
...
```

Remark:

KUSPY('ON'/'OFF' calls do not nest. If KUSPY('OFF') is called before KUSPY('ON') the behavior is undefined. KUSPY('OFF') can be called several times for a single call to KUSPY('ON').

4.2.2 Command Identification

CALL **KUPATH** (CHPATH*,NLEV*,NPAR*)

CALL **KUPATL** (CHPATL*,NPAR*)

char* path = **ku_path** ()

Parameter Description:

CHPATH	Array to return elements of full command path.
NLEV	Number of elements in CHPATH.
CHPATL	Bottom level path element (=CHPATH(NLEV)).
NPAR	Number of command line arguments (=KUNPAR()).
path	Complete command path.

Action: Retrieve the complete path of the command which invoked the action routine (KUPATH) or only the bottom level element (KUPATL).

KUPATH and KUPATL allow to handle several commands by a single action routine. For example, in the action routine for /HISTOGRAM/OPERATIONS/DIVIDE

```
CHARACTER*16 CHPATH(3),CHPATL
CALL KUPATH(CHPATH,NLEV,NPAR)
CALL KUPATL(CHPATL,NPAR)
```

will return

```
CHPATH(1) = 'HISTOGRAM'
CHPATH(2) = 'OPERATIONS'
CHPATH(3) = 'DIVIDE'
NLEV = 3
CHPATL = 'DIVIDE'
```

independent of the abbreviation used to invoke the command. CHPATH must be dimensioned to fit the maximum number of levels in the command path.

ku_path returns the complete command path, e.g. "/HISTOGRAM/OPERATIONS/DIVIDE". The functionality of KUPATH and KUPATL can be achieved easily enough using the C library string handling functions, for example

```
char *patl = strrchr( ku_path(), '/' ) + 1;
```

```
CALL KUHELP (LUNO*,CHPATL*)
```

Parameter Description:

LUNO	Logical unit number for output.
CHPATL	Bottom level command path element.

Action: Inquire logical unit for help text output.

In order to allow the redirection into a file the user help routines have to inquire the Fortran unit on which the additional help information has to be written. CHPATL allows to use a single user help routine for several commands. See page 90 for an example.

Remark:

There is no need for a C equivalent of KUHELP. See page 91 for an example of a user help routine in C.

```
CALL KUAPPL (LUNO*,INMACRO*,CHEXIT*)
```

```
char* exit = ku_appl (int* luno, int* inmacro)
```

Parameter Description:

LUNO	Logical unit number for application input.
INMACRO	Command/macro mode flag: 0 command line 1 macro
CHEXIT	Exit keyword.

Action: Return information about environment from which an "application" command was invoked.

KUAPPL must be called by commands which shall work in conjunction with the SET/APPLICATION command. There are two types of application commands: SIGMA-like and COMIS-like commands. SIGMA and COMIS are PAW commands, for which the concept of applications was originally invented.

The SIGMA command takes the complete argument line and interprets it as a vector expression. In that case

```
APPL SIGMA quit
v1=array(10,1#10)
v2=sqrt(v1)
quit
```

is simply an alternative[1] to

```
SIGMA v1=array(10,1#10)
SIGMA v2=sqrt(v1)
```

which comes in handy especially if a lot more than just two SIGMA commands have to be issued.

If SET/APPLICATION is used from the command line, the application command is called for each line of the application text, and the action routine has to react in the same way as if the command had been invoked explicitly. However, if SET/APPLICATION is used from within a macro, the application text is first written to a file, and the action routine is called only once. The skeleton for a SIGMA-like action routine is shown below:

[1] There is nevertheless a difference between these two forms: The "application text" is passed verbatim without any substitution of aliases etc.

Skeleton for a SIGMA-like action routine

```
      SUBROUTINE apsigma
      CHARACTER CHEXIT*32,CHTEMP*32,CHLINE*80

      CALL KUAPPL(LUNO,INMACRO,CHEXIT)
      IF(LUNO.EQ.5) THEN
        CALL KUGETE(CHLINE,NCH)
        CALL doit(CHLINE)
      ELSE
 1      CONTINUE
        CALL KUREAD(LUNO,CHLINE,NCH)
        IF(NCH.LT.0) THEN
          PRINT *,' Error reading application input'
        ELSE
          CHTEMP=CHLINE
          CALL CLTOU(CHTEMP)
          IF(CHTEMP.NE.CHEXIT) THEN
            CALL doit(CHLINE)
            GOTO 1
          ENDIF
        ENDIF
      ENDIF

      END
```

If `SET/APPLICATION` appears inside a macro, `KUAPPL` returns the Fortran unit from which the application text has to be read. The file is opened and closed by KUIP. The action routine has to check when the last input line has been reached by comparing it to `CHEXIT`. The comparison must be case-insensitive. Leading blanks are already stripped off by KUIP.

Remark:

It may be counterintuitive that the two case are distinguished by `LUNO` rather than by `INMACRO` but a test on `INMACRO` fails for an explicit "`SIGMA ...`" inside a macro. `INMACRO` is useful, for example, if the action routine wants to prompt the user for a correction in case of a mistake. On the other hand, if the command is called from within a macro there should not be any prompt because there might not be any user to reply!

`COMIS` is the prototype of commands which take over from KUIP the interactive dialog with the user. There "`APPL COMIS`" is useful only inside a macro in order to supply the input as in-line text.

Skeleton for a COMIS-like action routine

```
      SUBROUTINE apcomis
      CHARACTER CHEXIT*32,CHTEMP*32,CHLINE*80

      CALL KUAPPL(LUNO,INMACRO,CHEXIT)
      IF(INMACRO.EQ.0) THEN
        LUNO = 5
      ENDIF
```

```
1    CONTINUE
     IF(LUNO.EQ.5) THEN
       CALL KUINPS('Prompt >',CHLINE,NCH)
     ELSE
       CALL KUREAD(LUNO,CHLINE,NCH)
     ENDIF

     IF(NCH.LT.0) THEN
       PRINT *,' Error reading application input'
     ELSE
       CHTEMP=CHLINE
       CALL CLTOU(CHTEMP)
       IF(CHTEMP.NE.CHEXIT) THEN
         CALL doit(CHLINE)
         GOTO 1
       ENDIF
     ENDIF

     END
```

The massaging of LUNO is there only as a convenience for the user: in case he types "APPL COMIS" interactively it will be treated as the simple command COMIS.

Remark:

Inside a macro the other way around, i.e. "COMIS" instead of "APPL COMIS" cannot be used. In-line text is treated properly only if it is preceded by the SET/APPLICATION command.

4.2.3 Terminal Input/Output

When using the KUIP command loop (KUWHAT etc.) action routines should never issue a Fortran READ for keyboard input. Instead the following routines are provided to prompt the user for input of various type.

CALL **KUPROI** (CHPROMPT,*IVAL*)

CALL **KUPROR** (CHPROMPT,*RVAL*)

int value = **ku_proi** (const char* prompt, int dfault)

double value = **ku_pror** (const char* prompt, double dfault)

Parameter Description:

CHPROMPT	Prompt string.
IVAL,RVAL	Value entered by user.

Action: Prompt for an integer (KUPROI) or real (KUPROR) value. The prompt string includes the initial value of IVAL or RVAL as default proposal.

```
CALL KUPROS (CHPROMPT,*CHVAL*,LENGTH*)
```

```
CALL KUPROC (CHPROMPT,*CHVAL*,LENGTH*)
```

```
char* value = ku_pros (const char* prompt, const char* dfault)
```

```
char* value = ku_proc (const char* prompt, const char* dfault)
```

Parameter Description:

CHPROMPT	Prompt string.
CHVAL	Value entered by user.
LENGTH	Logical length of CHVAL.
dfault	Default value if not NULL.

Action: Prompt for a string value (KUPROS) possibly converted to uppercase (KUPROC). If the initial value of CHVAL is non-blank it is included in the prompt string as default proposal.

```
CALL KUPROF (CHPROMPT,*CHVAL*,LENGTH*)
```

```
CALL KUPROP (CHPROMPT,CHVAL*,LENGTH*)
```

```
char* value = ku_prof (const char* prompt, const char* dfault)
```

```
char* value = ku_prop (const char* prompt)
```

Parameter Description:

Same as for KUPROS.

Action: Prompt for a filename (KUPROF) or a password (KUPROP). KUPROF applies a case conversion as described for KUGETF. KUPROP does not echo the user input and provides no default.

```
CALL KUINPS (CHPROMPT,CHVAL*,LENGTH*)
```

```
char* value = ku_inps (const char* prompt)
```

Parameter Description:

Same as for KUPROS.

Action: In the basic KUIP command line interface KUINPS has the same functionality as KUPROS except the initial content of CHVAL is not used as default value. In KUIP/Motif KUPROS pops up a text window while KUINPS reads from the **"Executive Window"**.

Remark:

KUIP/Motif applications must use KUPROS or KUINPS instead of reading directly from the terminal.

`CALL` **KUALFA**

`void` **ku_alfa** `()`

Action: Call terminal routine defined by `KUTERM`.

In graphics applications each action routine has to call `KUALFA` before writing to the terminal in order to ensure that the device is in text mode.

`int answer =` **ku_more** `(const char* question, const char* text)`

Parameter Description:

`question,quote` Prompt string constructed from "*question* `"`*text*`" (Yes/No/Quit/Go) ?`". If `text` is `NULL` the quoted text is omitted.

`answer` Answer supplied by user:

1	Yes
2	No
3	Quit
4	Go

Action: Ask a question and wait for a 4-way answer.

`ISTAT =` **KUQKEY** `()`

`int status =` **ku_qkey** `()`

Parameter Description:

`ISTAT`	0	User did not hit the `RETURN` key
	1	User did hit the `RETURN` key

Action: Test if terminal input is available; do not wait if not.

`KUQKEY` allows to program quasi-infinite loops which the user can terminate by hitting the `RETURN` key:

```
      PRINT *,' Hit the RETURN key to stop'
    1 CONTINUE
      ...
      IF(KUQKEY().EQ.0) GOTO 1
```

4.2.4 Break handling

The `KUWHAT` command loops provide a break handler which allows to interrupt the current action routine and return to the command input prompt by hitting the keyboard interrupt key (usually `CTRL/C`). In addition an application may register (`KUBREK`) a routine to be called each time a keyboard interrupt is intercepted.

```
IOLD = KUBROF ()
```

```
CALL KUBRON
```

```
int old = ku_intr (int new)
```

Parameter Description:

`old`	Previous setting of break interrupt.
`new`	New setting of break interrupt.

Action: Disable (`KUBROF`) or enable (`KUBRON`) keyboard interrupts.

If an action routine contains a critical section, i.e. a sequence of statements which must never be interrupted in the middle, it should be protected by pairs of `KUBROF`/`KUBRON` calls. After calling `KUBROF` keyboard interrupts are delayed until they are enabled again by `KUBRON`.

KUIP calls `KUBRON` always before entering a new action routine in order to enabling the break handling, if previously disabled. Thus, explicit calls to `KUBRON` are only needed if the code between `KUBROF` and return from the action routine can take a sizeable amount of time that blocking the keyboard interrupt would jeopardize the interactive feel of the application.

If the critical section is in a subroutine which may be called from within another critical section the return value of `KUBROF` should be examined before calling `KUBRON`:

```
      SUBROUTINE CRITSEC
      ...
      IBRON = KUBROF()
*--- start of critical section
      ...
*--- end of critical section
      IF(IBRON.NE.0) THEN
*--- re-enable interrupts if they were enabled before
        CALL KUBRON
      ENDIF
      ...
```

Remark:

Note the difference between `KUBROF` and the command `SET/BREAK OFF`. `KUBROF` only delays the delivery of the keyboard interrupt. `SET/BREAK OFF` de-installs the signal handler that a keyboard interrupt will terminate the program.

`KUBROF` is equivalent to `ku_intr`(0) and `KUBRON` is equivalent to `ku_intr`(1).

```
CALL KUSIBR
```

```
void ku_sibr ()
```

Action: Simulate a break condition.

The break condition causes the signal handler to unwind the stack and to return to the command input loop.

Remark:

`KUSIBR` does not work on all platforms.

```
ISTOP = KUSTOP ()
```

```
int stop = ku_stop (int set)
```

Parameter Description:

ISTOP	Value of “soft interrupt” flag.
set	If non-zero, set soft interrupt flag, otherwise just return present value.

Action: Inquire soft interrupt flag.

An action routine looping over a possibly very large number of iterations should test KUSTOP in regular intervals and leave the loop in an orderly fashion:

```
      DO 10 I=1,N
        ...
        IF(KUSTOP().NE.0) GOTO 99
   10 CONTINUE
   99 CONTINUE
      PRINT ...
```

The soft interrupt flag can be set by sending the application program a SIGUSR2 signal.

4.3 KUIP Utilities

4.3.1 File editing

Filenames are of the form “fname.ftype” on all systems including IBM/VM-CMS. The file extension may be omitted and defaults to “.kumac”. Appending the default file extension can be inhibited by prepending the filename with a minus character.

```
LGSERV = KUQSVR ()
```

Action: Inquire whether edit server is active (see section 2.9.2).

```
CALL KUEDIT (CHFILE,ISTAT*)
```

```
CALL KUESVR (CHFILE,ISTAT*)
```

```
int status = ku_edit (const char* file, int use_server)
```

Parameter Description:

CHFILE Filename

ISTAT Return status from KUEDIT
- 0 file has been modified
- 1 file has not been modified
- 2 error

Return status from KUESVR:
- 0 file has been sent to edit server

	1	edit server not active
	2	error
status	Return status from ku_edit:	
	0	file has been modified
	1	file has not been modified or has been sent to edit server
	2	error
use_server	Send file to edit server if active.	

Action: Invoke the editor on the given file. The editor can be set with the HOST_EDITOR command.

KUEDIT starts a synchronous editing session, i.e. the application program waits until the editor terminates. KUESVR sends the file to the edit server and should only be called if KUQSVR returned .TRUE.

CALL **KUEUSR** (XTEDIT)

Parameter Description:

XTEDIT User edit routine.

Action: Define parameterless user routines called when an asynchronous editing session terminates. The user routine has to call KSVPAR to inquire the file name and the command name from which the editing session originated.

CALL **KSVPAR** (CHFILE*,CHPATH*)

Parameter Description:

CHFILE Filename
CHPATH Complete path of the command which invoked the edit server.

Action: Inquire file name and command name from the edit server was invoked.

Remarks:

KSVPAR should only be called in the user routine which has been registered by KUEUSR.

KUEUSR allows only to register a single user routine. If several commands want to use the edit server there should be a single routine using CHPATH to discriminate which action is required. If that is not possible, for example because the other user edit routine is in a library, chaining must be applied:

```
      EXTERNAL XTELIB
*-- user edit routine from library
      CALL KUEUSR(XTELIB)
      ...
      EXTERNAL XTEUSR
*-- user edit routine from user code overriding XTELIB
      CALL KUEUSR(XTEUSR)
      ...
      SUBROUTINE XTEUSR
      CHARACTER CHFILE*80,CHPATH*32
      CALL KSVPAR(CHFILE,CHPATH)
      IF(CHPATH.EQ.'/USER/EDIT') THEN
*-- handle this command only
```

```
         ...
       ELSE
*-- call next user edit routine
         CALL XTELIB
       ENDIF
       END
```

CALL **KUPAD** (CHFILE)

void **ku_pad** (const char* file, int is_temp)

Parameter Description:

CHFILE	Filename
is_temp	Flag if file is temporary and should be removed afterwards.

Action: Invoke the text browser on the given file. The browser can be set with the HOST_PAGER command.

4.3.2 Vector handling

CALL **KUVCRE** (CHNAME, CHTYPE, LENGTH, LLOW*, LHIGH*)

Action: creates a KUIP vector of a given type and length and returns its address in the ZEBRA store.

Parameter Description:

CHNAME	Vector name
CHTYPE	Vector type ('R' for real or 'I' for integer)
LENGTH	Vector length array (dimensioned to 3) with LENGTH(I) containing the I-th dimension length or 0 if the dimension is not used; e.g. LENGTH(1)=10 with LENGTH(2)=0 and LENGTH(3)=0 defines a one-dimensional vector of length 10.
LLOW	Returned low address in the Q store
LHIGH	Returned high address in the Q store

The vector is accessed as Q(LLOW:LHIGH) for CHTYPE='R' or as IQ(LLOW:LHIGH) for CHTYPE='I'. If LLOW=0 an error occurred.

CALL **KUVDEL** (CHNAME)

Action: deletes the KUIP vector named CHNAME. If CHNAME='*' all vectors are deleted.

Parameter Description:

CHNAME	Vector name

Example of the use of KUVECT

```
      COMMON /PAWC/ NWPAW,IXPAWC,IHBOOK,IXHIGZ,IXKUIP,IFENCE(5),
     +              LMAIN, WS(9989)
      DIMENSION IQ(1),Q(1),LQ(8000)
      EQUIVALENCE (Q(1),IQ(1),LQ(9)),(LQ(1),LMAIN)
      COMMON /QUEST/ IQUEST(100)

      CALL KUGETV('MYVECT',LLOW,LHIGH)

      IF (LLOW.EQ.0) THEN
        PRINT *,'Vector does not exist'
      ELSE
        IF (IQUEST(14).EQ.1) THEN
          PRINT *,'Element are:',(Q(I),I=LLOW,LHIGH)
        ELSE IF (IQUEST(14).EQ.2) THEN
          PRINT *,'Element are:',(IQ(I),I=LLOW,LHIGH)
        ENDIF
      ENDIF
```

CALL **KUVECT** (CHNAME, LLOW*,LHIGH*)

Action: gets address of vector CHNAME.

Parameter Description:

CHNAME	vector name
LLOW	low address, or 0 if the vector does not exist
LHIGH	high address, or 0 if the vector does not exist

Remark:

The vector CHNAME can be accessed by Q(LLOW:LHIGH) if ITYPE=1, or IQ(LLOW:LHIGH) if ITYPE=2 (Q and IQ are equivalenced in the ZEBRA store /PAWC/, see the example below). If the vector was not previously created then LLOW=LHIGH=0. The ZEBRA status vector IQUEST, which is generally used to pass information from ZEBRA to the user, will contain on return:

IQUEST(10)	NCHNAM, the number of characters of CHNAME.
IQUEST(11)	LENTOT, the total number of elements of the vector.
IQUEST(12)	ILOW, the lower boundary of the vector.
IQUEST(13)	IHIGH, the higher boundary of the vector.
IQUEST(14)	ITYPE, the type of the elements (1=real, 2=integer, 3=Hollerith)

CALL **KUVEC** (CHNAME, *X*, NELEMS,CHOPT)

Action: performs some vector operations, namely creation and copying of a Fortran array from/to a KUIP vector, depending on the argument CHOPT.

Parameter Description:

CHNAME	Vector name
X	Fortran array

`NELEMS`	Number of elements to be copied
`CHOPT`	Character variable specifying the required option

- `'R'` read into array `X` the content of vector `CHNAME` (i.e. `CHNAME => X`) starting at vector element `ILOW` (e.g. `ILOW=1` is the first one).
  `NELEMS` elements are read (`CHNAME(ILOW:ILOW+NELEMS-1)` or all if `NELEMS`≤`0` or `NELEMS`≥`LENTOT`.
  If the vector does not exist, then `IQUEST(1)=1`.
- `' '` same as `'R'`.
- `'W'` write array `X` into vector `CHNAME` (i.e. `X => CHNAME`) starting at vector element `ILOW` (e.g. `ILOW=1` is the first one). `NELEMS` elements are written e.g. `X(1:NELEMS)`. If the vector is not large enough, it is automatically extended. If the vector does not exist, it is created.
- `'-'` (used together with option `'W'`) same as `CHOPT='W'` and in addition the vector is shrunk to its actual size.
- `'C'` just create the vector if it does not exist.
- `'I'` (used together with option `'W'` or `'C'`) vector is of type Integer (default case is Real)
- `'H'` (used together with option `'W'` or `'C'`) vector is of type Hollerith (default case is Real)

Note that here `ILOW` stands for `ILOW(IDIM)`, `IHIGH` for `IHIGH(IDIM)` and `LENTOT` for `LENGTH(IDIM)`, where `IDIM` is the dimension (1, 2 or 3) affected.

IQUEST return code

`IQUEST(10)`	`NCHNAM` (number of characters of `CHNAME`)
`IQUEST(11)`	`LENTOT` (total number of elements of vector)
`IQUEST(12)`	`LLOW` (low address)
`IQUEST(13)`	`LHIGH` (high address)
`IQUEST(14)`	`ITYPE` (type: 1=real, 2=integer, 3=Hollerith)
`IQUEST(20)`	`ICOPY` (if <>0 a copy on a temporary vector was done, with `LENFR` and `LENTO` addresses defined as follow)
`IQUEST(21)`	`LENFR(1)`
`IQUEST(22)`	`LENFR(2)`
`IQUEST(23)`	`LENFR(3)`
`IQUEST(31)`	`LENTO(1)`
`IQUEST(32)`	`LENTO(2)`
`IQUEST(33)`	`LENTO(3)`

The vector elements can be addressed individually, if the common block `/PAWC/` is present, by `Q(LLOW+I)` or `IQ(LLOW+I)`, with `I` ranging from `0` to 1 `LENTOT-1`.

The array `X` should be defined in the calling routine of the right type, INTEGER or REAL, corresponding to the vector type.

Only one dimension can be handled by this routine, ex. `CHNAME` can be `VEC(3)`, `VEC(2,3:5)`, etc. if `VEC` is two-dimensional, but cannot be `VEC(2:3,3:5)`.

4.4 Stand-alone Utilities

4.4.1 File handling

```
CALL KUHOME (*CHFILE*,LENGTH*)
```

```
char* path = ku_home (const char* file, const char* ftype)
```

Parameter Description:

`CHFILE`	Filename to be expanded.
`LENGTH`	Logical length of expanded `CHFILE`.
`ftype`	Default filetype if `file` does not contain an extension.
`path`	Allocated string containing the expanded filename.

Action: Expand "~" and "$*var*" in Unix filenames.

```
LGCASE = KUQCAS ()
```

Action: Logical function returning the file case conversion:

`.TRUE.`	if `FILECASE CONVERT` is active.
`.FALSE.`	if `FILECASE KEEP` is active.

```
CALL KUFCAS (*CHFILE*)
```

```
char* case_file = ku_fcase (char *file)
```

Parameter Description:

`CHFILE`	Filename to be converted.
`case_file`	Return input pointer `file`.

Action: Apply file case conversion to input string.

```
CALL KUOPEN (LUNO,CHFILE,CHMODE,ISTAT*)
```

```
int status = ku_open (int luno, const char* file, const char* mode)
```

Parameter Description:

`LUNO`	Logical unit number	
`CHFILE`	File name in the form "`fname.ftype`"	
`CHMODE`	File status:	
	`'OLD'`	Open existing file for read-only access. Return error if the file does not exist.
	`'NEW'`	Create new file and open it for write access. Return error if the file already exists, except for VMS where a new cycle is created.

'UNKNOWN'	Like 'NEW' but return no error if the file does already exist. On VMS a new cycle is created.
'APPEND'	Like 'UNKNOWN' but the write pointer is positioned at EOF if the file does already exist. On VMS the highest existing cycle is opened.
'DONTKNOW'	Like 'UNKNOWN' except for VMS where the highest existing cycle is opened. Use if it is not known whether the file will be read or written.
'VERYOLD'	Like 'OLD' except for VM/CMS (see below) where file must be RECFM~F.
'VERYNEW'	Like 'UNKNOWN' except for VM/CMS where file is created with RECFM~F and LRECL~80.

ISTAT Return status zero if open succeeded.

Action: Open a file on Fortran unit LUNO for FORMATTED access.

Remarks:

KUOPEN does not necessarily match the behavior of an OPEN statement with the same STATUS=CHMODE value.

I/O of lines longer than 80 characters are not supported on all systems.

```
CALL KUREAD (LUNO,CHLINE*,LENGTH*)
```

```
CALL KUWRIT (LUNO,CHLINE)
```

```
int length = ku_read (int luno, char* line, size_t len)
```

```
void ku_write0 (int luno, const char* line)
```

Parameter Description:

LUNO	Logical unit number
CHLINE	Character buffer
LENGTH	Logical length of CHLINE or -1 when end of file has been reached
len	Maximum number of characters to read. The input buffer must dimensioned as char line[len+1].

Action: Read/write one line to/from Fortran unit.

Remark:

The VM/CMS Fortran runtime system is not capable of formatted I/O operations on RECFM~V files. In order to perform formatted I/O there are two possibilities:

– Use mode 'VERYOLD' and Fortran formatted READ statements. KUOPEN will report an error if the file is not RECFM~F:

```
      CALL KUOPEN(LUNO,CHFILE,'VERYOLD',ISTAT)
      IF(ISTAT.NE.0) THEN
        PRINT *,' File does not exist or is not RECFM F'
        GOTO 999
      ENDIF
1     CONTINUE
      READ(LUNO,1000,END=2,ERR=99) ...
1000  FORMAT(...)
```

```
      ...
      GOTO 1
2     CONTINUE
      ...
```

– Use mode 'OLD', read from the file into a CHARACTER variable and from there into the final destination:

```
      CHARACTER CHLINE*80
      ...
      CALL KUOPEN(LUNO,CHFILE,'OLD',ISTAT)
      IF(ISTAT.NE.0) THEN
        PRINT *,' File does not exist'
        GOTO 999
      ENDIF
1     CONTINUE
      CALL KUREAD(LUNO,CHLINE,NCH)
      IF(NCH.GE.0) THEN
        READ(CHLINE,1000,END=99,ERR=99) ...
1000    FORMAT(...)
        ...
        GOTO 1
      ENDIF
      ...
```

`CALL` **KUCLOS** `(LUNO,CHMODE,ISTAT*)`

`int status =` **ku_close** `(int luno)`

Parameter Description:

LUNO	Logical unit number	
CHMODE	File status:	
	' '	Close unit and leave file.
	'DELETE'	Close unit and delete file. Cannot be applied if file has been opened with mode 'OLD'.
ISTAT	Return status zero if close successful.	

Action: Close file connected to Fortran logical unit.

`CALL` **KUINQF** `(*CHFILE*,*LUNO*)`

`int luno =` **ku_inqf** `(const char* file)`

Parameter Description:

CHFILE<>' ' inquires the existence and open status of file CHFILE and returns in LUNO:

-1 if the file does not exist,

	`0`	if the file exists but is not open,
	n	if the file is opened as logical unit *n*.

`CHFILE=' '` inquires the open status of unit `LUNO` and returns the name of the connected file in `CHFILE` or leaves `CHNAME=' '` if the unit is not used.

Return codes

`IQUEST(11)` returns the format mode :

- `1` `FORMATTED`
- `2` `UNFORMATTED`
- `0` other

`IQUEST(12)` returns the access type :

- `1` `SEQUENTIAL`
- `2` `DIRECT`
- `0` other

`IQUEST(13)` returns the record length of direct access files.

Action: Inquire about open Fortran files and their corresponding logical units.

4.4.2 Program arguments

```
CALL KGETAR (CHARGS*)
```

```
char* args = k_getar ()
```

Parameter Description:

`CHARGS` Command line arguments.

Action: Return command line arguments used to invoke the program.

The arguments are separated by single blanks. The program name itself is not included. The system function `$ARGS` (see page 35) returns the same information.

```
void k_setar (size_t argc, char** argv)
```

Parameter Description:

`argc` First argument of C `main` function.
`argv` Second argument of C `main` function.

Action: Keep command line arguments if main program is written in C.

The Fortran runtime library provide usually a routine such as `GETARG` to retrieve the command line arguments for Fortran main programs. For C main programs, however, the command line arguments have to be passed to `k_setar` or they will not be accessible later on.

Linking with a C main program will result in two unresolved externals `f77argc` and `f77argv` (or similar names depending on the system). These are the global variables referenced by `GETARG` where the Fortran main program would store the command line arguments. The C main program can provide dummy definitions for them in order to avoid the error messages at link time:

```
int f77argc;
char** f77argv;

main( int argc, char** argv )
{
  ...
  k_setar( argc, argv );
  ...
}
```

Remark:
On some systems mixing of Fortran and C works only if the main program is in Fortran, e.g. because the Fortran runtime library needs special initializations. Check the implications for all possible target platforms before you depend on writing the main program in C!

CALL **KUARGS** (CHPROG,*CHLOGON*,CHBATCH*,CHLOGF*,IERROR*)

Parameter Description:

CHPROG program name used in error messages

CHLOGON logon macro file name (initial value unchanged unless overruled by command line arguments)

CHBATCH batch macro file name

CHLOGF log file name (same as the batch file but with the extension .LOG)

IERROR error code (0 = no error)

Action: Parse standard command line arguments.
Each KUIP application should provide two facilities:

- Execute a logon macro at startup time in order to allow the user to customize the working environment.
- Execute a batch macro without further interaction with the user.

KUARGS parses the arguments with which the application program was invoked in order to allow the user:

- Inhibit the execution of the logon macro.
- Specify a different name for the logon macro.
- Specify the name of the batch macro.

The command line syntax depends on the operating system and follows the customary rules for specifying program options:

	Unix	VMS	VM/CMS
Inhibit logon macro	-n	/NOLOG	(NOLOG
Change logon macro	-l *name*	/LOGON=*name*	(LOGON=*name*
Execute batch macro	-b *name*	/BATCH=*name*	(BATCH=*name*

4.4.3 Conversion functions

```
CALL KUDPAR (CHSTRING,IPAR*,RPAR*,CHPAR*,LENGTH*,CHTYPE*)
```

Parameter Description:

`CHSTRING`	string containing the value of unknown type
`IPAR`	integer return value if `CHTYPE='I'`
`RPAR`	real return value if `CHTYPE='R'`
`CHPAR`	character return value if `CHTYPE='C'`
`LENGTH`	logical length of `CHPAR`
`CHTYPE`	type of value given in `CHSTRING`

- `'I'` for INTEGER An integer parameter is defined as a string readable by a Fortran internal READ using the `I` format descriptor.
- `'R'` for REAL A real parameter is defined as a string readable by a Fortran internal READ using `F` or `E` format descriptors.
- `'C'` for CHARACTER A character parameter is defined as a string not readable by a Fortran internal READ using any of the `I`, `F` or `E` format descriptors.
- `' '` in case of error

Action: Determine type of value contained in string.

```
int matches = ku_match (const char* string, const char* pattern,
                        int ignore_case)
```

Action: Match `string` against `pattern` containing "*" as wild-cards matching any sequence of characters. If `ignore_case` is non-zero, the comparison is case-insensitive.

4.4.4 String handling

The functions described in the following are prototyped in the `/cern/pro/include/kstring.h` header file. This file can be used as a replacement of `#include <string.h>`.

```
void* dst = memmove (void* dst, const void* src, size_t n)
```

Action: Like `memcpy` but source and destination may be overlapping.

```
int cmp = strcasecmp (const char* str1, const char* str2)
```

```
int cmp = strncasecmp (const char* str1, const char* str2, size_t n)
```

Action: Like `strcmp` and `strncmp` but comparison is case insensitive.

`char* pos =` **strrstr** `(const char* str1, const char* str2)`

Action: Like `strstr` but searching for last occurrence.

`char* pos =` **strrpbrk** `(const char* str1, const char* str2)`

Action: Like `strpbrk` but searching for last occurrence.

`char* token =` **strqtok** `(char* str)`

Action: Like `strtok(str," ")` but ignoring blanks protected by quotes.

`char* str =` **strlower** `(char* str)`

`char* str =` **strupper** `(char* str)`

Action: Convert string to lower/upper case.

`char* str =` **strtrim** `(char* str)`

Action: Remove leading and trailing blanks.

`char* ptr =` **struntab** `(char* ptr)`

Action: Replace TAB characters by equivalent number of blanks, assuming tab stops every 8th character. The input string must be allocated and the return string is possibly relocated.

`char* str =` **strfromd** `(double value, size_t prec)`

`char* str =` **strfromi** `(int value, size_t prec)`

Action: `strfromd` and `strfromi` return a pointer to a static buffer containing the string representation of `value`.

For `strfromd` the `prec` argument determines the number of significant digits. `prec=0` uses an appropriate default.

For `strfromi` the `prec` argument allows to specify the field width filled with leading zeroes. `prec=0` uses just the width necessary to represent the value.

`double value =` **fstrtod** `(const char* str, char** tail)`

`int value =` **fstrtoi** `(const char* str, char** tail)`

Action: Like `strtod/strtol` but `if(*tail=='\0')` test if the complete string forms a valid number.

```
char* ptr =  strdup   (const char* str)
```

```
char* ptr =  str0dup  (const char* str)
```

```
char* ptr =  strndup  (const char* str, size_t n)
```

Action: `strdup` returns pointer to allocated copy of input string (which must not be `NULL`).
`str0dup` returns `NULL` if the input string is NULL, empty, or contains only blanks.
`strndup` uses only the first `n` characters of the input string.

```
char* ptr =  str2dup  (const char* str1, const char* str2)
```

```
char* ptr =  str3dup  (const char* str1, const char* str2,
                       const char* str3)
```

```
char* ptr =  str4dup  (const char* str1, const char* str2,
                       const char* str3, const char* str4)
```

```
char* ptr =  str5dup  (const char* str1, const char* str2,
                       const char* str3, const char* str4,
                       const char* str5)
```

Action: Return pointer to allocated concatenation of input strings.

```
char* ptr =  stridup  (int number)
```

Action: Return pointer to allocated string representation of `number`.

```
char* ptr =  mstrcat  (char* ptr, const char* str)
```

```
char* ptr =  mstr2cat  (char* ptr, const char* str1, const char* str2)
```

```
char* ptr =  mstr3cat  (char* ptr, const char* str1, const char* str2,
                        const char* str3)
```

```
char* ptr =  mstr4cat  (char* ptr, const char* str1, const char* str2,
                        const char* str3, const char* str4)
```

Action: Concatenate strings to allocated pointer and return possibly relocated pointer.

```
char* ptr =  mstrncat  (char* ptr, const char* str, size_t n)
```

Action: Concatenate the first `n` characters of string.

`char* ptr =` **mstrccat** `(char* ptr, char c, size_t n)`

Action: Concatenate `n` times the character `c`.

`char* ptr =` **mstricat** `(char* ptr, int number)`

Action: Concatenate the string representation of `number`.

`int status =` **shsystem** `(const char* shell, const char* command)`

Action: Like `system` but pass `command` to the given `shell`.

`char* file =` **fexpand** `(const char* fname, const char* ftype)`

`char* file =` **fsearch** `(const char* fname, const char* ftype, const char* path)`

Action: Expand "~" and environment variables in file name. If `ftype` is not `NULL` add default file extension.

`fsearch` searches for the file in the comma-separated directory list in `path` and returns the expanded file name or `NULL` if no file is found.

`char* ptr =` **fstrdup** `(const char* buf, size_t len)`

`char* ptr =` **fstr0dup** `(const char* buf, size_t len)`

Action: Return allocated copy of input buffer ignoring trailing blanks. `fstr0dup` returns `NULL` if buffer consists of blanks only.

`char* ptr =` **fstrtrim** `(const char* buf, size_t len)`

`char* ptr =` **fstr0trim** `(const char* buf, size_t len)`

Action: Return allocated copy of input buffer ignoring leading and trailing blanks. `fstr0trim` returns `NULL` if buffer consists of blanks only.

`size_t n =` **fstrlen** `(const char* buf, size_t len)`

Action: Return logical length of buffer ignoring trailing blanks.

`size_t n =` **fstrset** `(char* buf, size_t len, const char* str)`

Action: Copy string into buffer padded with blanks and return logical length.

`char* buf =` **fstrvec** `(char** pstr, size_t n, size_t* len)`

Parameter Description:

`pstr`	String array
`n`	Number of elements in `pstr`. If `n=0` then `pstr` is `NULL` terminated.
`len`	Return length of each item in buffer.

Action: Convert C string array to Fortran `CHARACTER` array.

`size_t len =` **mstrlen** `(char** pstr, size_t n)`

Parameter Description:

`pstr`	String array
`n`	Number of elements in `pstr`. If `n=0` then `pstr` is `NULL` terminated.

Action: Return maximum length of strings in `pstr` rounded up to next multiple of machine word size.

4.4.5 Hash tables

The functions described in the following are prototyped in the `/cern/pro/include/khash.h` header file. They provide a simple method for manipulating symbol tables and other cases where a data structure has to be associated with a name.

The hash table functions in the form presented here are available only since the 95a release.

`HashTable* table =` **hash_create** `(int size)`

Action: Create a hash table which is organized as an array of `size` linked lists. A hashing algorithm determines the index of the linked list in which a given name should be located.

Therefore `size` does not limit the number of entries which can be added to the table. It only allows to select the trade-off between the amount of memory occupied by an empty table and the average search time. Folklore has it that the hash indices are most evenly distributed if `size` is a prime number.

`void` **hash_config** `(HashTable* table, const char* option)`

Action: Configure the behavior of the hash table.

If needed, `hash_config` must be called whilst the table is still empty.

`"string"`	(default) The values stored are character strings which are copied into allocated memory automatically.
`"struct"`	The values are structure pointers. It is the programmers responsibility to free allocated memory when deleting entries.
`"tag_only"`	Only the `tag` field is used. The `value` field is ignored.
`"keep"`	(default) The `name` given to `hash_insert` is stored in its original spelling.
`"lower"`	The `name` given to `hash_insert` is stored converted to lowercase.
`"upper"`	The `name` given to `hash_insert` is stored converted to uppercase.
`"ignore"`	(default) Name lookups are case-insensitive.
`"respect"`	Name lookups are case-sensitive.

```
void  hash_insert  (HashTable* table, const char* name,
                    const void* value, int tag)
```

Action: Enter an entry to the hash table.

An already existing entry with the same `name` is replaced by the new definitions. The `name` is stored in its original spelling but the table lookups are case-insensitive.

If the hash option `"string"` is set, then `value` is supposed to be a string which is automatically copied into allocated memory. If the hash option `"struct"` is used, then it is the programmer's responsibility that the structure pointed to by `value` stays intact until it is removed again from the table.

```
void* value =  hash_lookup  (HashTable* table, const char* name,
                             int* tag_return)
```

Action: Retrieve an entry from the hash table.

`NULL` is returned if an entry with the given `name` does not exist. The `tag` field is written into `tag_return` if it is not `NULL`.

```
int entries =  hash_entries  (HashTable* table)
```

Action: Return the number of entries stored in the hash table.

```
HashArray* array =  hash_array  (HashTable* table)
```

Action: Return all entries in an array of `entries` elements.

The elements are sorted by name in (case-insensitive) alphabetical order. If the table is empty, `hash_array` returns `NULL`. Otherwise the pointer should be `free`d afterwards.

Example for printing hash table content

```
int n = hash_entries( table );

if( n == 0 ) {
  printf( " *** No entries.\n" );
}
else {
  HashArray* array = hash_array( table );
  int i;

  for( i = 0; i < n; n++ ) {
    int number = i + 1;
    char *name = array[i].name;
    char *value = (char*)array[i].value;
    int flag = array[i].tag;

    printf( "%d: %s = %s / %d\n", number, name, value, flag );
  }
  free( (char*)array );
}
```

`void* value =` **hash_remove** `(HashTable* table, const char* name)`

Action: Delete an entry from the hash table.

For option `"string"` the memory allocated to hold the string value is released automatically and `hash_remove` returns always `NULL`. Otherwise the `value` given to `hash_insert` is returned and it is the programmers responsibility to perform any necessary cleanup operation on that pointer.

It is harmless to try removing a non-existing entry.

`void` **hash_clear** `(HashTable* table)`

Action: Delete all entries from the hash table.

In order to avoid memory leaks `hash_clear` should only be used for option `"struct"` tables if the `value` pointers are static. See the example below how to clear the table if the pointers are allocated.

`void` **hash_destroy** `(HashTable* table)`

Action: Call `hash_clear` and then release the memory allocated for the hash table itself.

Examples:

The following examples illustrate the use of the hash functions when storing string, integer, or floating point values. The last example can be easily generalized to storing more complex structures than just a `double` value. In each example we perform the following operations:

– create the table
– insert two entries
– remove one entry
– lookup an entry and print its value if it exists
– remove all entries from the table

Symbol table with string values

```
HashTable* table = hash_create( 97 );

hash_insert( table, "name1", "1", 0 );
hash_insert( table, "name2", "2", 0 );

hash_remove( table, "name2" );

char* value;
if( (value = (char*)hash_lookup( table, "name1", NULL )) != NULL ) {
  printf( " name1 = %s\n", value );
}

hash_clear( table );
```

Symbol table with integer values

```
HashTable* table = hash_create( 97 );
hash_config( table, "tag_only" );

hash_insert( table, "name1", NULL, 1 );
hash_insert( table, "name2", NULL, 2 );

hash_remove( table, "name2" );

int value;
if( hash_lookup( table, "name1", &value ) != NULL ) {
  printf( " name1 = %d\n", value );
}

hash_clear( table );
```

Symbol table with double values

```
HashTable* table = hash_create( 97 );
hash_config( table, "struct" );

double* dval;
dval = (double*)malloc( sizeof(double) );
*dval = 1.0;
hash_insert( table, "name1", dval, 0 );
dval = (double*)malloc( sizeof(double) );
*dval = 2.0;
hash_insert( table, "name2", dval, 0 );

if( (dval = (double*)hash_remove( table, "name2" )) != NULL ) {
  free( (char*)dval );
}

if( (dval = hash_lookup( table, "name1", NULL )) != NULL ) {
  double value = *dval;
  printf( " name1 = %f\n", value );
}

/* hash_clear() would not free the memory */
int n = hash_entries( table );
if( n > 0 ) {
  HashArray* array = hash_array( table );
  int i;
  for( i = 0; i < n; i++ ) {
    free( (char*)array[i].value );
  }
  free( (char*)array );
}
```

4.5 Main Program Skeletons

Following are examples of a user main program using KUIP.

User main program for basic KUIP

```
      PROGRAM MAIN
*
      COMMON/PAWC/PAW(50000)
      EXTERNAL MYEXIT
      CALL INITC
*
* Initialize ZEBRA and the store /PAWC/
*
      CALL MZEBRA(-3)
      CALL MZPAW(50000,' ')
*
* Initialize KUIP with NWORDS words as minimum division size
*
      NWORDS=5000
      CALL KUINIT(NWORDS)
*
* Create the user command structure from the definition file
* generated automatically by the KUIP Compiler.
* The command definition routine name has been defined in the
* CDF (Command Definition File) through the control line '>Name DEF'
*
      CALL DEF
*
* Define the exit routine.
* Entering the command 'EXIT' this routine is executed,
* then the program continues from the line after CALL KUWHAT (or KUWHAG).
*
      CALL KUEXIT(MYEXIT)
*
* Change the prompt
*
      CALL KUEXEC('SET/PROMPT ''My_prompt >''')
*
* Execute a logon macro
*
      CALL KUEXEC('EXEC MYLOGON')
*
* Give control to KUIP (with no 'STYLE G', call KUWHAG to have 'STYLE G').
* Entering the command 'QUIT' or 'EXIT' the program continues
* from the line after CALL KUWHAT (or CALL KUWHAG).
*
      CALL KUWHAT
*
* Entering the command 'QUIT' or 'EXIT' we return here
*
      END
```

User main program for KUIP/Motif without HIGZ graphics

```
      PROGRAM MAIN
*
* Basic KUIP/Motif Application (no HIGZ graphics)
*
      COMMON/PAWC/PAW(50000)
      CALL INITC
*
* Initialize ZEBRA and the store /PAWC/
*
      CALL MZEBRA(-3)
      CALL MZPAW(50000,' ')
*
* Initialize KUIP with NWORDS words as minimum division size
*
      NWORDS=2000
      CALL KUINIT(NWORDS)
*
* Create the user command structure from definition file (CDF)
*
      CALL VECDEF
*     CALL ...
*
* Gives access to KUIP browsers for commands, files and macros
*
      CALL KUIDFM
*
* Execute some KUIP Initialization Commands
*
      CALL KUEXEC('PROMPT ''KTEST >''')
      CALL KUEXEC('LAST 0')
*
* Give control to MOTIF
*
      CALL KUWHAM ('Ktest')
*
      END
```

If your application requires high level graphics managed by HIGZ, then you have to add a small part for the HIGZ (and eventually HPLOT) initialization. The following is a user main program skeleton for a graphical application using HIGZ and HPLOT.

User main program for KUIP/Motif with HIGZ graphics

```
      PROGRAM MAIN
*
* Basic KUIP/Motif Application (no HIGZ graphics)
*
      PARAMETER (NWHIGZ=10000)
      COMMON/PAWC/PAW(50000)
*
      EXTERNAL      IGTERM
```

```
      CALL INITC
*
* Initialize ZEBRA and the store /PAWC/
*
      CALL MZEBRA(-3)
      CALL MZPAW(50000,' ')
*
* Initialize KUIP with NWORDS words as minimum division size
*
      NWORDS=2000
      CALL KUINIT(NWORDS)
*
* Create the user command structure from definition file (CDF)
*
      CALL VECDEF
      CALL KTDEF            ! (*) see below ...
*     CALL ...              ! (code generated from the CDF compilation)
*
* Gives access to KUIP browsers for commands, files and macros
*
      CALL KUIDFM
*
* Special KUIP initialization for using Motif with HIGZ
*
      CALL KUINIM('Ktest')
*
* Initialize HIGZ
*
      CALL IGINIT(NWHIGZ)
      CALL KUGRFL(IGTERM)   ! flush the graphics output after each command
*
* Initialize HPLOT
*
      IWK=999
      CALL HPLINT(IWK)
      CALL IGSA(0)
*
* Execute some KUIP Initialization Commands
*
      CALL KUEXEC('PROMPT ''KTEST >''')
      CALL KUEXEC('LAST 0')
*
* Give control to MOTIF
*
      CALL KUWHAM ('Ktest')
*
      END
```

(*) The subroutine KTDEF is automatically generated from the CDF (directive ``>Name KTDEF''. This CDF must contain the directive "`>Graphics`" (see sections 3.4.5.1 and 3.7.1).

5 Built-in commands

5.1 Menu KUIP

Command Processor commands.

`KUIP/HELP [ item option ]`

`ITEM`	C	"Command or menu path"	`D='␣'`
`OPTION`	C	"View mode"	`D='N'`

Possible `OPTION` values are:

`EDIT`	The help text is written to a file and the editor is invoked,
`E`	Same as 'EDIT'.
`NOEDIT`	The help text is output on the terminal output.
`N`	Same as 'NOEDIT'

Give the help of a command. If ITEM is a command its full explanation is given: syntax (as given by the command USAGE), functionality, list of parameters with their attributes (prompt, type, default, range, etc.). If ITEM='/' the help for all commands is given.

If HELP is entered without parameters or ITEM is a submenu, the dialogue style is switched to 'AN', guiding the user in traversing the tree command structure.

'HELP -EDIT' (or just 'HELP -E') switches to edit mode: instead of writing the help text to the terminal output, it is written into a temporary file and the pager or editor defined by the command HOST_PAGER is invoked. (On Unix workstations the pager can be defined to display the help text asynchrously in a separated window.) 'HELP -NOEDIT' (or just 'HELP -N') switches back to standard mode. The startup value is system dependent.

`KUIP/USAGE item`

`ITEM`	C	"Command name"

Give the syntax of a command. If ITEM='/' the syntax of all commands is given.

KUIP/MANUAL item [output option]

ITEM	C	"Command or menu path"
OUTPUT	C	"Output file name" D='␣'
OPTION	C	"Text formatting system" D='␣'

Possible OPTION values are:

'␣'	plain text : plain text format
LATEX	LaTeX format (encapsulated)
TEX	LaTeX format (without header)

Write on a file the text formatted help of a command. If ITEM is a menu path the help for all commands linked to that menu is written. If ITEM='/' the help for the complete command tree is written. If OUTPUT=' ' the text is written to the terminal.

The output file produced with option LATEX can be processed directly by LaTeX, i.e. it contains a standard header defining the meta commands used for formatting the document body. With option TEX only the document body is written into the output file which can be included by a driver file containing customized definitions of the standard meta commands. Example:

```
MANUAL / MAN.TEX LATEX
```

will produce the file MAN.TEX containg the documentation of all available commands in LaTeX format.

KUIP/EDIT fname

FNAME	C	"File name"

Invoke the editor on the file. The command HOST_EDITOR can be used to define the editor.

If FNAME does not contain an extension the default filetype '.KUMAC' is supplied. The search path defined by the command DEFAULTS is used to find an already existing file. If the file does not exist it is created with the given name.

KUIP/PRINT fname

FNAME	C	"File name"

Send a file to the printer. The command HOST_PRINT can be used to define the host command for printing the file depending on it file extension.

KUIP/PSVIEW fname

`FNAME` C "File name"

Invoke the PostScript viewer on the file. The command HOST_PSVIEWER can be used to define the PostScript viewer.

If FNAME does not contain an extension the default filetype '.PS' is supplied.

KUIP/LAST [n fname]

`N` I "N last commands to be saved" `D=-99 R=-99:`

`FNAME` C "File name" `D='␣'`

Perform various operations with the history file.

If FNAME is not specified, the current history file is assumed by default (the startup history file name is LAST.KUMAC). To change the history file the command LAST 0 NEW-FNAME must be entered.

If N.EQ.-99 (default case) the default host editor is called to edit the current history file, containing all the commands of the session.

If N.LT.0 the last -N commands are printed on the screen. On MVS this allows to edit and resubmit commands. On workstations this allows to resubmit blocks of commands by mouse-driven cut-and-paste operations.

If N.EQ.0 the history file FNAME is rewound and set as the current one (the command LAST 0 FNAME itself is not recorded).

If N.GT.0 the last N commands of the session are saved in the current history file.

See also the command RECORDING.

KUIP/MESSAGE [string]

`STRING` C "Message string" `D='␣' Separate`

Write a message string on the terminal. A useful command inside a macro. Several message strings can be given in the same command line, each of them separated by one or more spaces (the usual parameter separator); therefore multiple blanks will be dropped and only one will be kept. If multiple blanks should not be dropped, the string must be surrounded by single quotes.

KUIP/SHELL [cmd]

`CMD` C "Shell command string" `D='␣'`

Execute a command of the host operating system. The command string is passed to the command processor defined by HOST_SHELL. If CMD=' ' the shell is spawned as interactive subprocess. To return from the shell enter 'RETURN' (the full word, not just ⟨CR⟩) or 'exit' (depending on the operation system).

KUIP/WAIT [string sec]

STRING	C	"Message string" D='␣'
SEC	R	"Number of seconds" D=0 R=0:

Make a pause (e.g. inside a macro). Wait a given number of seconds (if SEC.GT.0) or just until ⟨CR⟩ is entered (if SEC.EQ.0). A message string is also written on the terminal before waiting.

KUIP/IDLE sec [string]

SEC	I	"Number of seconds" R=0:
STRING	C	"Command string" D='␣'

Execute a command if program is idle. The command string is executed if there was no keyboard activity during SEC seconds.

KUIP/UNITS

List all Input/Output logical units currently open. The files attached to them are also shown.

KUIP/EXIT

End of the interactive session.

KUIP/QUIT

End of the interactive session.

KUIP/FUNCTIONS

```
        *** KUIP System Functions ***
```

The function name (and arguments) is literally replaced, at run-time, by its current value. At present, the following functions are available:

```
$DATE  ....................... Current date in format DD/MM/YY
$TIME  ....................... Current time in format HH.MM.SS
$CPTIME  ..................... CP time elapsed since last call (in sec)
$RTIME  ...................... Real time elapsed since last call (in sec)
$VDIM(VNAME,IDIM)  ........... Physical length of vector VNAME
                               on dimension IDIM (1..3)
```

```
$VLEN(VNAME,IDIM) ........... As above, but for the logical length
                              (i.e. stripping trailing zeroes)
$NUMVEC ..................... Current number of vectors
$VEXIST(VNAME) .............. Index of vector VNAME
                              (1..$NUMVEC or 0 if VNAME does not exist)
$SUBSTRING(STRING,IX,NCH) ... STRING(IX:IX+NCH-1)
$UPPER(STRING) .............. STRING changed to upper case
$LOWER(STRING) .............. STRING changed to lower case
$LEN(STRING) ................ Length of STRING
$INDEX(STR1,STR2) ........... Position of first occurrence of STR2 in
STR1
$WORDS(STRING,SEP) .......... Number of words separated by SEP
$WORD(STRING,K,N,SEP) ....... Extract N words starting at word K
$QUOTE(STRING) .............. Add quotes around STRING
$UNQUOTE(STRING) ............ Remove quotes around STRING
$EXEC('macro args') ......... EXITM value of EXEC call
$DEFINED('var_name') ........ List of defined macro variables
$EVAL(Expression) ........... Result of the Expression computed by KUIP
$SIGMA(Expression) .......... Result of the Expression computed by SIGMA
$RSIGMA(Expression) ......... As above but a decimal point is added to
                              integer results
$FORMAT(number,format) ...... Format a number according to a Fortran
                              format string, e.g.
                              $FORMAT(1.5,F5.2) ==> ' 1.50'
                              $FORMAT(123,I5.5) ==> '00123'
$ARGS ....................... Command line at program invocation
$KEYNUM ..................... Address of latest clicked key in style GP
$KEYVAL ..................... Value of latest clicked key in style GP
$LAST ....................... Latest command line executed
$ANUM ....................... Number of aliases
$ANAM(I) .................... Name of I-th alias
$AVAL(I) .................... Value of I-th alias
$STYLE ...................... Current style as defined by SET/STYLE
$OS ......................... Operating system name, e.g. UNIX or VMS
$MACHINE .................... Hardware or Unix brand, e.g. VAX or HPUX
$PID ........................ Process ID
$IQUEST(I) .................. Value of IQUEST(I) status vector
$ENV(var) ................... Value of environment variable
$FEXIST(file) ............... 1 if file exists or 0 otherwise
$SHELL(cmd,N) ............... N'th line of shell command output (Unix
only)
$SHELL(cmd,sep) ............. Shell output with newlines replaced by sep
$SHELL(cmd) ................. Same as $SHELL(cmd,' ')
```

5.2 Menu KUIP/ALIAS

Operations with aliases. Aliases are defined to provide shortcut abbreviations for the input line or some part of it. When encountered on an input line an alias is replaced by its string value which can contain further aliases. (Be careful not to define recursive aliases.)

To juxtaposition aliases, a double slash can be used as concatenation sign. Inside quoted strings and for the ALIAS commands themselves the alias substitution is inhibited. Otherwise

```
ALIAS/CREATE ALPHA BETA
ALIAS/CREATE ALPHA BETA
```

whould create an recursive alias BETA and

```
ALIAS/CREATE ALPHA BETA
ALIAS/CREATE BETA GAMMA
ALIAS/DELETE ALPHA
```

would delete the alias name BETA instead of ALPHA itself.

KUIP/ALIAS/CREATE name value [chopt]

NAME	C	"Alias name"
VALUE	C	"Alias value"
CHOPT	C	"Option" D='A'

Possible CHOPT values are:

- A create an Argument alias
- C create a Command alias
- N No alias expansion of value

Create an alias NAME which should be substituted by VALUE. An alias name is a sequence of letters and digits starting with a letter. The underscores ('_'), the at-sign ('@') and the dollar-sign ('$') count as letters.

There are two types of aliases: Command aliases are recognized only if they occur in the command position, i.e. as the first token on the line. Argument aliases are recognized anywhere on the command line (except inside quoted strings) if they are surrounded by one of the following separators:

```
blank  /  ,  =  :  .  %  '  (  )
```

Also switch ON the alias translation, i.e. ALIAS/TRANSLATION ON. If CHOPT='C' then the alias is a command alias, i.e. an alias that will only be translated when it is the first token on a command line. Example:

```
Alias/Create GG Graph/Struct/Scratch
Alias/Create FF File1/Name1/Name2
GG FF/ID
```

is equivalent to

```
Graph/Struct/Scratch File1/Name1/Name2/ID
```

```
Alias/Create LS DIR C
```

is equivalent to

```
DIR
```

only when LS is the first token on a command line. In the following case LS will not be translated

```
SHELL LS
```

Aliases occuring inside an value are expanded indepedent whether the value is enclosed by quotes. The option -N allows to suppress this implicit alias expansion.

KUIP/ALIAS/LIST [name]

`NAME` C "Alias name wildcard" `D='*'`

List all aliases matching the wildcard (names and values).

KUIP/ALIAS/DELETE name

`NAME` C "Alias name wildcard" `Loop`

Delete the definition of aliases matching the wildcard. NAME='*' deletes all aliases.

KUIP/ALIAS/TRANSLATION [option]

`OPTION` C "Option" `D='ON'`

Possible `OPTION` values are:

`?`	show current setting
`ON`	switch alias translation ON
`OFF`	switch alias translation OFF

Switch ON/OFF the alias translation. If OFF, alias definitions are not used in parsing the command lines. It is automatically switched ON when an alias is created. If OPTION='?' the current value is shown. The startup value is OFF.

5.3 Menu KUIP/SET_SHOW

Set or show various KUIP parameters and options.

KUIP/SET_SHOW/STYLE [option sgylen sgsize sgyspa sgbord wktype]

OPTION	C	"Option" D='?'
SGYLEN	R	"max Y LENgth of each menu item box" D=0.025 R=0.005:0.25
SGSIZE	R	"space available for the application" D=0.8 R=0:0.90
SGYSPA	R	"max Y length of space between menus" D=0.02 R=-0.5:0.50
SGBORD	R	"X or Y border for menus" D=0.015 R=0:0.25
WKTYPE	I	"Graphics workstation type" D=0

Possible OPTION values are:

?	show current style
C	Command line : select Command line input
AN	Menu with Numbers : select general Alpha menu (with Numbers)
AL	Menu with Letters : select general Alpha menu (with Letters)
G	Graphics menu hardware : select Graphics menu (with hardware character fonts)
GW	Graphics menu shadowed : select Graphics menu (with shadowed Width effect)
GS	Graphics menu Software : select Graphics menu (with Software character fonts)
GP	Panel keys : select Graphics menu (with Panel keys only, i.e. no command tree menu)
XM	Motif/X11 : select Motif/X11 interface

Select the user dialogue style (or working mode). The startup value is 'C' (command mode). The current value is returned by the system function $STYLE.

The G-styles are only available if the application program is calling KUWHAG instead of KUWHAT. When one of these options is choosen the remaining parameters control the geometrical layout of the menus on the screen and the graphics workstation type (in case HIGZ was not initialized).

Style 'XM' is only available if the program is calling KUWHAM. In that case switching to other styles is not possible.

KUIP/SET_SHOW/NEWPANEL line col title width height xpos ypos

LINE	I	"Number of lines" D=5 R=1:30
COL	I	"Number of columns" D=5 R=1:30
TITLE	C	"Panel Title" D='New Panel'
WIDTH	I	"Panel width (in pixels)" D=300 R=10:
HEIGHT	I	"Panel height (in pixels)" D=300 R=10:
XPOS	I	"X Position (in pixels)" D=0 R=0:
YPOS	I	"Y Position (in pixels)" D=0 R=0:

Set up a new panel with empty keys (to be filled interactively).

KUIP/SET_SHOW/COMMAND [chpath]

`CHPATH` C "Path name for command line" `D='␣'`

Set a filter for the parsing of command lines. If it has been called, it means that whenever a command line is entered, if and only if it is not an existing command (not just ambiguous), it is inserted into the CHPATH string, with $n (n=1..9) being replaced by the n-th token of the command (tokens are separated by spaces), or $* being replaced by the whole command line. Examples:

```
COMMAND 'V/CR $*(10)'
AA                      =>   V/CR AA(10)
BB                      =>   V/CR BB(10)
V/LIST                  =>   V/LIST

COMMAND 'VECTOR/PLOT $1 555 $2'
AA E                    =>   VECTOR/PLOT AA 555 E
BB                      =>   VECTOR/PLOT BB 555

COMMAND                 =>   shows its current value
COMMAND *               =>   reset (equivalent to COMMAND $*)
```

Note that COMMAND and subsequent command lines can be used inside macros, excepted when producing macro statements (like EXEC, IF, GOTO, etc.). For example, the above examples would work also inside macros, while COMMAND 'EXEC $*' or COMMAND 'GOTO $1' will not.

KUIP/SET_SHOW/APPLICATION path [cmdex]

`PATH` C "Application name" `D='␣'`
`CMDEX` C "Exit command" `D='EXIT'`

Set the application name. This means that all input lines will be concatenated to the string PATH (until the command specified by the parameter CMDEX is executed, which resets the application to the null string). The value of CMDEX may be specified if the default value EXIT has to be changed (i.e. because already used by the application). APPLICATION can also be inserted in a macro: in this case at least 4 characters must be specified (i.e. APPL).

KUIP/SET_SHOW/ROOT [path]

`PATH` C "Root directory" `D='/'`

Set the root for searching commands. If PATH='?' the current root is shown. This allows to access commands regardless of possible ambiguities with different menus. Commands are first searched starting from the current root: if a command is found it is executed. Only if a command is not found a second pass of search is done, starting now from the top root of the command tree (i.e. '/').

KUIP/SET_SHOW/TIMING [option]

`OPTION` C "Option" `D='ON'`

Possible `OPTION` values are:

`ON`
`OFF`
`ALL`

Set ON/OFF/ALL the timing of commands. If ON, the real time and the CPU time for the latest executed command (or macro) are presented. If ALL, the time is shown for each command being executed within a macro. The startup value is OFF.

KUIP/SET_SHOW/PROMPT prompt

`PROMPT` C "Prompt string" `D='␣'`

Set the prompt string for the command mode dialogue. If PROMPT is blank the current prompt is left unchanged. If PROMPT contains the character sequence '[]' the current command number is inserted between the square brackets.

KUIP/SET_SHOW/BREAK [option]

`OPTION` C "Option" `D='ON'`

Possible `OPTION` values are:

`ON`
`OFF`
`TB`
`?`

Set ON/OFF the break handling. If OPTION='?' the current value is shown. The startup value is ON.

Hitting the keyboard interrupt (CTRL/C on VMS or CTRL/Q on the Apollo) under break ON condition, the current command or macro execution will be interrupted and the user will get again the application prompt.

BREAK TB switch ON the traceback of the routines called, with their line numbers, when an error occurs. This allows the detection of the routines which provoked the error.

KUIP/SET_SHOW/COLUMNS [ncol]

`NCOL` I "Number of columns for terminal output" `D=80 R=-1:`

Set the maximum number of columns for terminal output. If NCOL=0 the current number of columns is shown. If NCOL=-1 the current number of columns is taken from the environment variable COLUMNS. If COLUMNS is undefined the startup value is 80.

KUIP/SET_SHOW/RECORDING [nrec]

`NREC` I "Rate for recording on history file" `D=25 R=0:`

Set the recording rate for the history file. Every NREC commands of the session the current history file is updated. If NREC=0 the history is not kept at all (i.e. the file is not written). See also the command LAST.

KUIP/SET_SHOW/HOST_EDITOR [editor top left width height dxpad dypad npads]

`EDITOR`	C	"Host editor command" `D='?'`
`TOP`	I	"Top position of the edit window" `D=20 R=0:`
`LEFT`	I	"Left position of the edit window" `D=20 R=0:`
`WIDTH`	I	"Width of the edit window" `D=0 R=0:`
`HEIGHT`	I	"Height of the edit window" `D=0 R=0:`
`DXPAD`	I	"X offset for help PAD windows" `D=30 R=0:`
`DYPAD`	I	"Y offset for help PAD windows" `D=20 R=0:`
`NPADS`	I	"Maximum number of shifted pads" `D=4 R=1:`

Set the host command to invoke the editor. The EDIT command will invoke this editor. If EDITOR='?' the current host editor command is shown.

On Apollo the special value EDITOR='DM' invoke Display Manager pads. The special values EDITOR='WINDOW' and 'PAD' can be used to specify the window positions (in pixel units). 'WINDOW' defines the parameters for edit pads, while 'PAD' defines the parameters for read-only pads (e.g. used by 'HELP -EDIT').

On VMS the special values EDITOR='EDT' and 'TPU' invoke the callable editors. The startup time is considerably lower compared to spawning the editor as a subprocess. The callable EDT has one disadvantage though: after an error, e.g. trying to edit a file in a non-existing directory, subsequent calls will always fail. The TPU call can be augmented by command line options, e.g.

```
HOST_EDITOR TPU/DISP=DECW    | DECwindow interface to EVE
```

On Unix a variety of editors are available, e.g.

```
HOST_EDITOR vi
HOST_EDITOR 'emacs -geometry 80x48'
```

On Unix workstations it is possible to do asynchronous editing via the KUIP edit server, i.e. to start an editor in a separate window while the application can continue to receive commands. In order to do that the following conditions must be fulfilled:

```
- The KUIP edit server 'kuesvr' must be found in the search path.
- The editor command set by HOST_EDITOR must end with an ampersand ('&').
- The environment variable 'DISPLAY' must be set.
```

The ampersand flags your intention to use the edit server if possible. If the edit server cannot be used the ampersand will be ignored, i.e. even with

```
HOST_EDITOR 'vi &'
```

the KUIP/EDIT command will block until the editor terminates if either the 'kuesvr' is not available or 'DISPLAY' is undefined. When using the edit server the editor command is expected to create its own window. 'vi' being a frequent choice, the above command is automatically interpreted as

```
HOST_EDITOR 'xterm -e vi &'
```

The startup value can be defined by the environment variable 'EDITOR'. Otherwise it is set to a system dependent default: 'DM' (Apollo), 'EDT' (VMS), 'XEDIT' (VM/CMS), 'vi' (Unix).

KUIP/SET_SHOW/HOST_PAGER [pager]

`PAGER` C "Host pager command" `D='?'`

Set the host command to view a file in read-only mode. If OPTION='?' the current host pager command is shown. The 'HELP -EDIT' command will invoke this pager, e.g.

```
HOST_PAGER more
```

On Unix workstations the pager can be asynchronous by creating a separate window, e.g.

```
HOST_PAGER 'xterm -e view &'
HOST_PAGER 'ved &'
```

On Apollo the special value PAGER='DM' defines the use of Display Manager read-only pads. The pad positions can be adjusted by the HOST_EDITOR command.

The startup value can be defined by the environment variables 'KUIPPAGER' or 'PAGER'. If neither of them is defined the value set by the HOST_EDITOR command is used. On VAX/VMS the startup value is 'TYPE/PAGE'.

KUIP/SET_SHOW/HOST_PRINTER [command filetype]

`COMMAND` C "Host printer command" `D='?'`

`FILETYPE` C "File extension" `D='␣'`

Set the host commands for printing files with KUIP/PRINT. The KUIP/PRINT command will use the host command matching the file extension or use the default command defined for FILETYPE=' '.

If COMMAND='?' the currently set commands are shown. If COMMAND=' ' the currently defined command is delete. The command string can contain '$*' and '$-' to indicate the position where the file name with/without file extension should be inserted. For example,

```
MANUAL / refman.tex latex
HOST_PRINTER 'latex $* ; dvips $-' .tex
KUIP/PRINT refman.tex
```

invokes the shell command 'latex refman.tex ; dvips refman'. The predefined defaults are not guaranteed to work since the actual print commands are very much installation dependent.

KUIP/SET_SHOW/HOST_PSVIEWER [psviewer]

`PSVIEWER` C "Host PostScript Viewer command" `D='?'`

Set the host command to invoke the PostScript Viewer. The PSVIEW command will invoke this PostScript Viewer. If PSVIEWER='?' then the current viewer command is shown.

The startup value can be defined by the environment variables 'KUIPPSVIEWER' or 'PSVIEWER'.

On Unix workstations it is by default set to 'ghostview'. On VAX/VMS the default commands is 'VIEW/FORM=PS/INTERFACE=DECWINDOWS'.

KUIP/SET_SHOW/HOST_SHELL [shell]

`SHELL` C "Host shell command" `D='?'`

Set the default host shell invoked by the KUIP/SHELL command. If OPTION='?' the current host shell is shown. The startup value is taken from the 'SHELL' environment variable.

KUIP/SET_SHOW/RECALL_STYLE [option]

`OPTION` C "Command recall and editing style" `D='?'`

Possible `OPTION` values are:

`?`	show current setting
`KSH`	Korn shell : Emacs like command line editing
`KSHO`	Korn shell + Overwrite : like 'KSH' but overwrite instead of insert mode
`DCL`	VAX/VMS DCL : DCL command line editing
`DCLO`	VAX/VMS DCL + Overwrite : like 'DCL' but overwrite instead of insert mode
`NONE`	disable command line editing

Set the command recall and editing style. If OPTION='?' the current style is shown. The startup value is 'DCL' on VAX/VMS, 'NONE' on Cray and Apollo DM pads, and 'KSH' on other systems.

If the terminal emulator returns ANSI escape sequences (hpterm doesn't!) the up/down arrow keys can be used to recall items from the command history list and the left/right arrow keys to move the cursor.

'KSH' style provides the following control keys for editing:

```
^A/^E    : Move cursor to beginning/end of the line.
^F/^B    : Move cursor forward/backward one character.
^D       : Delete the character under the cursor.
^H, DEL  : Delete the character to the left of the cursor.
^K       : Kill from the cursor to the end of line.
^L       : Redraw current line.
^O       : Toggle overwrite/insert mode. Text added in overwrite mode
           (including yanks) overwrites existing text, while insert mode
           does not overwrite.
^P/^N    : Move to previous/next item on history list.
^R/^S    : Perform incremental reverse/forward search for string on
           the history list.  Typing normal characters adds to the
           current search string and searches for a match.  Typing
           ^R/^S marks the start of a new search, and moves on to
           the next match.  Typing ^H or DEL deletes the last
           character from the search string, and searches from the
           starting location of the last search.
           Therefore, repeated DELs appear to unwind to the match
           nearest the point at which the last ^R or ^S was typed.
           If DEL is repeated until the search string is empty the
           search location begins from the start of the history
           list. Typing ESC or any other editing character accepts
           the current match and loads it into the buffer,
           terminating the search.
^T       : Toggle the characters under and to the left of the cursor.
^U       : Kill from the prompt to the end of line.
^Y       : Yank previously killed text back at current location.
           Note that this will overwrite or insert, depending on
           the current mode.
TAB      : By default adds spaces to buffer to get to next TAB stop
           (just after every 8th column).
LF, CR   : Returns current buffer to the program.
```

'DCL' style provides the following control keys for editing:

```
BS/^E    : Move cursor to beginning/end of the line.
^F/^D    : Move cursor forward/backward one character.
DEL      : Delete the character to the left of the cursor.
^A       : Toggle overwrite/insert mode.
^B       : Move to previous item on history list.
^U       : Delete from the beginning of the line to the cursor.
```

```
TAB     : Move to next TAB stop.
LF, CR  : Returns current buffer to the program.
```

KUIP/SET_SHOW/VISIBILITY cmd [chopt]

CMD C "Command name" D='␣'
CHOPT C "?, OFF, ON" D='?'

Possible CHOPT values are:

?
OFF
ON

Set or show the visibility attributes of a command.
If CHOPT='OFF':

```
- the command it is not executable anymore
- STYLE G draws a shadowed box on the command
- HELP may be still requested on the command
```

The startup value is ON.

KUIP/SET_SHOW/DOLLAR [option]

OPTION C "Substitution of environment variables" D='?'

Possible OPTION values are:

?	show current setting
ON	enable substitution
OFF	disable substitution

Set or show the status of environment variable substitution.

This command allows to enable/disable the interpretation of environment variables in command lines. The startup value is 'ON', i.e. "$var" is substituted by the variable value.

Note that the system function "$ENV(var)" allows using environment variables even for 'DOLLAR OFF' .

KUIP/SET_SHOW/FILECASE [option]

OPTION C "Case conversion for filenames" D='?'

Possible OPTION values are:

?	show current setting
KEEP	filenames are kept as entered on the command line
CONVERT	filenames are case converted
RESTORE	restore previous FILECASE setting

Set or show the case conversion for filenames.

This command has only an effect on Unix systems to select whether filenames are kept as entered on the command line. The startup value is 'CONVERT', i.e. filenames are converted to lowercase.

On other systems filenames are always converted to uppercase.

The 'RESTORE' option set the conversion mode to the value effective before the last FILECASE KEEP/CONVERT command. E.g. the sequence

```
FILECASE KEEP; EDIT Read.Me; FILECASE RESTORE
```

forces case sensitivity for the EDIT command and restores the previous mode afterwards.

KUIP/SET_SHOW/LCDIR [directory]

DIR*ECTORY C "Directory name" D='␣'

Set or show the local working directory.

The current working directory is set to the given path name or the current directory is shown.

To show the current directory used LCDIR without argument. 'LCDIR ' switches to the home directory. 'LCDIR .' switches back to the working directory at the time the program was started.

5.4 Menu MACRO

Macro Processor commands.

MACRO/EXEC mname [margs]

MNAME C "Macro name"
MARGS C "Macro arguments" D='␣' Separate

Execute the command lines contained in the macro MNAME. As a file can contain several macros, the character '#' is used to select a particular macro inside a file as explained below.

If MNAME does not contain the character '#', the file MNAME.KUMAC is searched and the first macro is executed (it may be an unnamed macro if a MACRO statement is not found as first command line in the file).

If MNAME is of the form FILE#MACRO, the file named FILE.KUMAC is searched and the macro named MACRO is executed.

Examples:

```
EXEC ABC   to exec first (or unnamed) macro of file ABC.KUMAC
EXEC ABC#M to exec macro M of file ABC.KUMAC
```

The command MACRO/DEFAULTS can be used to define a directory search path for macro files.

MACRO/LIST [mname]

MNAME C "Macro name pattern" D='␣'

List all macros in the search path defined by MACRO/DEFAULTS. Macros are files with the extension KUMAC. MNAME may be specified to restrict the list to the macros containing such a string in the first part of their name. For example,

```
MACRO/LIST ABC
```

will list only macros starting with ABC.

MACRO/TRACE [option level]

OPTION C "Option" D='ON'
LEVEL C "Level" D='␣'

Possible `OPTION` values are:

```
ON
OFF
```

Possible `LEVEL` values are:

```
'␣'
TEST
WAIT
FULL
DEBUG
```

Set ON/OFF the trace of commands during macro execution. If TRACE='ON' the next command is written on the terminal before being executed. If LEVEL='TEST' the command is only echoed but not executed. If LEVEL='WAIT' the command WAIT is automatically inserted after the execution of each command. The startup values are OPTION='OFF' and LEVEL=' '.

MACRO/DEFAULTS [path option]

`PATH`	C	"Search path for macro files"	`D='?'`
`OPTION`	C	"Automatic EXEC"	`D='?'`

Possible `OPTION` values are:

`?`	show current setting
`Command`	search for commands only
`C`	same as 'Command'
`Auto`	search for commands before macros
`A`	same as 'Auto'
`AutoReverse`	search for macros before commands
`AR`	same as 'AutoReverse'

Set or show MACRO search attributes.

On Unix and VMS systems PATH defines a comma separated list of directories in which the commands KUIP/EDIT, MACRO/EXEC, and MACRO/LIST search for macro files. For example,

```
MACRO/DEFAULT '.,macro,~/macro'          | Unix
MACRO/DEFAULT '[],[.macro],[macro]'      | VMS
```

defines to search files first in the current directory, then in the subdirectory 'macro' of the current directory, and last the subdirectory 'macro' of the home directory.

On VM/CMS system PATH defines a comma separated list of filemodes. E.g.

```
MACRO/DEFAULT '*'       | search all disks
MACRO/DEFAULT 'A,C'     | search only disks A and C
```

If PATH='?' the currently defined search path is shown. If PATH='.' the search path is undefined, i.e. files are search for in the current directory (A-disk on VM/CMS) only. The startup value is PATH='.'.

The search path is not applied if the file specification already contains an explicit directory path or if it starts with a '-' character (which is stripped off).

OPTION allows to define whether macros can be invoked by their name only without prepending the KUIP/EXEC command:

```
DEFAULT -Command
CMD                     | CMD must be a command
DEFAULT -Auto
CMD                     | if CMD is not a command try EXEC CMD
DEFAULT -AutoReverse
CMD                     | try EXEC CMD first; if not found try command CMD
```

The startup value is 'Command' (also reset by PATH='.').

Important note:

Inside macros the DEFAULT -A (or -AR) logic is disabled, i.e. DEFAULT -C is always assumed.

MACRO/DATA

Application command to store immediate data into a file. Example:

```
Application DATA vec.dat
1  2  3
4  5  6
7  8  9
vec.dat
vec/read x,y,z vec.dat
```

5.5 Menu MACRO/GLOBAL

Operations on global variables.

MACRO/GLOBAL/CREATE name [value text]

`NAME` C "Variable name" `Loop`
`VALUE` C "Initial value" `D='␣'`
`TEXT` C "Comment text" `D='␣'`

Create a global variable.

If used inside a macro the variable [name] is declared as global.

MACRO/GLOBAL/IMPORT name

`NAME` C "Variable name" `Loop`

Import global variables.

If used inside a macro the variables listed are declared as global. The name may contain '*' as a wildcard matching any sequence of characters.

MACRO/GLOBAL/DELETE name

`NAME` C "Variable name" `Loop`

Delete global variables.

The global variables listed are deleted. The name may contain '*' as a wildcard matching any sequence of characters.

MACRO/GLOBAL/LIST [name file]

`NAME` C "Variable name" `D='*'`
`FILE` C "Output file" `D='␣'`

List global variables.

If a file name is specified the output is the list of GLOBAL/CREATE commands to define the selected global variables. The default file extension is .kumac.

5.6 Menu MACRO/SYNTAX

Explanation of KUIP macro syntax.

A macro is a set of command lines stored in a file, which can be created and modified with any text editor.

In addition to all available KUIP commands the special "macro statements" listed below are valid only inside macros. Note that the statement keywords are fixed. Aliasing such as "ALIAS/CREATE jump GOTO" is not allowed.

5.7 Menu MACRO/SYNTAX/Expressions

Explanation of KUIP expression syntax.

KUIP has a built-in parser for different kinds of expressions: arithmetic expressions, boolean expressions, string expressions, and "garbage expressions".

MACRO/SYNTAX/Expressions/Arithmetic

Explanation of arithmetic expression syntax.

The syntactic elements for building arithmetic expressions are:

```
expr ::=  number
        | vector-name                (for scalar vectors)
        | vector-name(expr)
        | vector-name(expr,expr)
        | vector-name(expr,expr,expr)
        | [variable-name]            (if value is numeric or
                                      the name of a scalar vector)
        | [variable-name](expr...)   (if value is a vector name)
        | alias-name                 (if value is numeric constant)
        | $system-function(...)
        | - expr
        | expr + expr
        | expr - expr
        | expr * expr
        | expr / expr
        | (expr)
        | ABS(expr)
        | INT(expr)
        | MOD(expr,expr)
```

They can be used in the macro statements DO, FOR, and EXITM, in macro variable assignments, as system function arguments where a numeric value is expected, or as the argument to the $EVAL function.

Note that all arithmetic operations are done in floating point, i.e., "5/2" becomes "2.5". If a floating point result appears in a place where an integer is expected, for example as an index, the value is truncated.

MACRO/SYNTAX/Expressions/Boolean

Explanation of Boolean expression syntax.

Boolean expressions can only be used in the macro statements IF, WHILE, and REPEAT. The possible syntactic elements are shown below.

```
 bool  ::= expr rel-op expr
         | string eq-op string
         | expr eq-op string
         | .NOT. bool
         | bool .AND. bool
         | bool .OR. bool
         | ( bool )

rel-op ::= .LT. | .LE. | .GT. | .GE.
         |  <   |  <=  |  >   |  >=
         | eq-op

eq-op  ::= .EQ. | .NE.
         |  =   | <>
```

MACRO/SYNTAX/Expressions/String

Explanation of string expression syntax.

String expressions can be used in the macro statements CASE, FOR, and EXITM, in macro variable assignments, as system function arguments where a string value is expected, or as the argument to the $EVAL function. They may be constructed from the syntactic elements shown below.

```
string ::= quoted-string
         | unquoted-string
         | string // string          (concatenation)
         | expr // string            (expr represented as string)
         | [variable-name]
         | alias-name
         | $system-function(...)
```

MACRO/SYNTAX/Expressions/Garbage

Explanation of "garbage" expression syntax.

Expressions which do not satisfy any of the other syntax rules we want to call "garbage" expressions. For example,

```
s = $OS$MACHINE
```

is not a proper string expression. Unless they appear in a macro statement where specifically only an arithmetic or a boolean expression is allowed, KUIP does not complain about these syntax errors. Instead the following transformations are applied:

```
o  alias substitution

o  macro variable replacement; values containing a
   blank character are implicitly quoted

o  system function calls are replaced one by one with
   their value provided that the argument is a syntactically
   correct expression

o  string concatenation
```

5.8 Menu MACRO/SYNTAX/Variables

Explanation of KUIP macro variables.

Macro variables do not have to be declared. They become defined by an assignment statement,

```
name = expression
```

The right-hand side of the assignment can be an arithmetic expression, a string expression, or a garbage expression (see MACRO/SYNTAX/Expressions). The expression is evaluated and the result is stored as a string (even for arithmetic expressions).

A variable value can be used in other expressions or in command lines by enclosing the name in square brackets, [name]. If the name enclosed in brackets is not a macro variable then no substitution takes place.

MACRO/SYNTAX/Variables/Numbered

Accessing macro arguments.

The EXEC command can pass arguments to a macro. The arguments are assigned to the numbered variables [1], [2], etc., in the order given in the EXEC command. The name of the macro, including the file specification, is assigned to [0].

A numbered variable cannot be redefined, i.e., an assignment such as "1 = foo" is illegal. See MACRO/SYNTAX/SHIFT.

MACRO/SYNTAX/Variables/Special

Predefined special macro variables.

For each macro the following special variables are always defined:

```
[0]     Fully qualified name of the macro.
[#]     Number of macro arguments
[*]     List of all macro arguments, separated by blanks
[@]     EXITM return code of the last macro called by
        the current one.  The value is "0" if the last
        macro did not supply a return code or no macro
        has been called yet.
```

As for numbered variables these names cannot be used on the left-hand side of an assignment. The values or [#] and [*] are updated by the SHIFT statement.

MACRO/SYNTAX/Variables/Indirection

Referencing a macro variable indirectly.

Macro variables can be referenced indirectly. If the variable [name] contains the name of another variable the construct

```
[%name]
```

is substituted by that other variable's value. For example, this is another way to traverse the list of macro arguments:

```
DO i=1,[#]
  arg = [%i]
  ...
ENDDO
```

There is only one level of indirection, i.e., the name contained in "name" may not start with another "%".

MACRO/SYNTAX/Variables/Global

Declaring a global variable.

```
EXTERN name ...
```

The variable names listed in the EXTERN statement are declared as global variables. If a name has not been defined with the GLOBAL/CREATE command, it is created implicitly and initialized to the empty string. The name list may contain wildcards, for example

```
EXTERN *
```

makes all defined global variables visible.

MACRO/SYNTAX/Variables/READ

Reading a variable value from the keyboard.

```
READ name  [ prompt ]
```

Variable values can be queried from the user during macro execution. The READ statement prompts for the variable value. If name is already defined the present value will be proposed as default.

MACRO/SYNTAX/Variables/SHIFT

Manipulation numbered variables.

The only possible manipulation of numbered variables is provided by the SHIFT statement which copies [2] into [1], [3] into [2], etc., and discards the value of the last defined numbered variable. For example, the construct

```
WHILE [1] <> ' ' DO
  arg = [1]
  ...
  SHIFT
ENDDO
```

allows to traverse the list of macro arguments.

5.9 Menu MACRO/SYNTAX/Definitions

Statements for defining macros.

MACRO/SYNTAX/Definitions/MACRO

Defining a macro.

A .kumac file may contain several macros. An individual macro has the form

```
MACRO macro-name [ parameter-list ]
   statements
RETURN
```

Each statement is either a command line or one of the macro constructs described in this section (MACRO/SYNTAX). For the first macro in the file the MACRO header can be omitted. For the last macro in the file the RETURN trailer may be omitted. Therefore a .kumac file containing only commands (like the LAST.KUMAC) already constitutes a valid macro.

MACRO/SYNTAX/Definitions/RETURN

Ending a macro definition

```
RETURN [ value ]
```

The RETURN statement flags the end of the macro definition and not the end of macro execution, i.e., the construct

```
IF ... THEN
  RETURN         | error!
ENDIF
```

is illegal. See MACRO/SYNTAX/EXITM.

The value is stored into the variable [@] in the calling macro. If no value is given it defaults to zero.

MACRO/SYNTAX/Definitions/EXITM

Terminate macro execution and return to calling macro.

```
EXITM [ value ]
```

In order to return from a macro prematurely the EXITM statement must be used. The value is stored into the variable [@] in the calling macro. If no value is given it defaults to zero.

MACRO/SYNTAX/Definitions/STOPM

Terminate macro execution and return to command line prompt.

```
STOPM
```

The STOPM statement unwinds nested macro calls and returns to the command line prompt.

MACRO/SYNTAX/Definitions/ENDKUMAC

Ignore rest of KUMAC file.

A logical "end of file" marker. The KUIP parser will not read any part of a .kumac file which appears after the "ENDKUMAC" command.

5.10 Menu MACRO/SYNTAX/Branching

Macro statements for general flow control.

MACRO/SYNTAX/Branching/CASE

Select one of many branches.

```
CASE expression IN
(label)  statement  [ statements ]
...
(label)  statement  [ statements ]
ENDCASE
```

The CASE switch evaluates the string expression and compares it one by one against the label lists until the first match is found. If a match is found the statements up to the next label are executed before skipping to the statement following the ENDCASE. None of the statements are executed if there is no match with any label.

Each label is a string constant and the comparison witht the selection expression is case-sensitive. If the same statement sequence should be executed for distinct values a comma-separated list of values can be used.

The "*" character in a label item acts as wildcard matching any string of zero or more characters, i.e., "(*)" constitutes the default label.

MACRO/SYNTAX/Branching/GOTO_and_IF_GOTO

Unconditional and conditional branching.

```
GOTO label
```

The simplest form of flow control is provided by the GOTO statement which continues execution at the statement following the target "label:". If the jump leads into the scope of a block statement, for example a DO-loop, the result is undefined.

The target may be given by a variable containing the actual label name.

```
IF expression GOTO label
```

This old-fashioned construct is equivalent to

```
IF expression THEN
   GOTO label
ENDIF
```

MACRO/SYNTAX/Branching/IF_THEN

Conditional execution of statement blocks.

```
IF expression THEN
   statements
ELSEIF expression THEN
   statements
...
ELSEIF expression THEN
   statements
ELSE
   statements
ENDIF
```

The general IF construct executes the statements following the first IF/ELSEIF clause for with the boolean expression is true and then continues at the statement following the ENDIF. The ELSEIF clause can be repeated any number of times or can be omitted altogether. If none of the expressions is true, the statements following the optional ELSE clause are executed.

MACRO/SYNTAX/Branching/ON_ERROR

Installing an error handler.

Each command returns a status code which should be zero if the operation was successful or non-zero if any kind of error condition occurred. The status code can be tested by $IQUEST(1) system function.

```
ON ERROR GOTO label
```

installs an error handler which tests the status code after each command and branches to the given label when a non-zero value is found. The error handler is local to each macro.

```
ON ERROR EXITM  [ expression ]
```

and

```
ON ERROR STOPM
```

are short-hand notations for a corresponding EXITM or STOPM statement at the targat label.

```
ON ERROR CONTINUE
```

continues execution with the next command independent of the status code. This is the initial setting when entering a macro.

```
OFF ERROR
```

An error handler can be deactivated by this statement.

```
ON ERROR
```

An error handler can be reactivated by this statement.

5.11 Menu MACRO/SYNTAX/Looping

Macro statements for construction loops.

MACRO/SYNTAX/Looping/DO

Loop incrementing a loop counter.

```
DO loop = start_expr, finish_expr  [, step_expr ]
   statements
ENDDO
```

The step size (setp_expr) defaults to "1". The arithmetic expressions involved can be floating point values but care must be taken of rounding errors.

Note that "DO i=1,0" results in zero iterations and that the expressions are evaluated only once.

MACRO/SYNTAX/Looping/FOR

Loop over items in an expression list.

```
FOR name IN expr_1 [ expr_2 ... expr_n ]
   statements
ENDFOR
```

In a FOR-loop the number of iterations is determined by the number of items in the blank-separated expression list. The expression list must not be empty. One by one each expression evaluated and assigned to the variable name before the statements are executed.

The expressions can be of any type: arithmetic, string, or garbage expressions, and they do not need to be all of the same type. In general each expression is a single list item even if the result contains blanks.

The variable [*] is treated as a special case being equivalent to the expression list "[1] [2] ... [n]" which allows yet another construct to traverse the macro arguments:

```
FOR arg IN [*]
   ...
ENDFOR
```

MACRO/SYNTAX/Looping/REPEAT

Loop until condition becomes true.

```
REPEAT
   statements
UNTIL expression
```

The body of a REPEAT-loop is executed at least once and iterated until the boolean expression evaluates to true.

MACRO/SYNTAX/Looping/WHILE

Loop while condition is true.

```
WHILE expression DO
   statements
ENDWHILE
```

The WHILE-loop is iterated while the boolean expression evaluates to true. The loop body is not executed at all if the boolean expression is false already in the beginning.

MACRO/SYNTAX/Looping/BREAKL

Terminate a loop.

```
BREAKL [ level ]
```

Allows to terminate a loop prematurely. The BREAKL continues executing after the end clause of a DO, FOR, WHILE, or REPEAT block, where "level" indicates how many nested constructs to terminate. The default value level=1 terminates the innermost loop construct.

MACRO/SYNTAX/Looping/NEXTL

Continue with next loop iteration.

```
NEXTL [ level ]
```

Allows to continue with the next loop iteration without executing the rest of the loop body. Execution continues just before the end clause of a DO, FOR, WHILE, or REPEAT block, where "level" indicates how many nested blocks to skip. The default value level=1 skips to the end of the innermost loop construct.

5.12 Menu VECTOR

Vector Processor commands. Vectors are equivalent to FORTRAN 77 arrays and they use the same notation except when omitting indexes (see last line below). Up to 3 dimensions are supported. Examples:

```
Vec(20) (mono-dimensional with 20 elements)
```

may be addressed by:

```
Vec         for all elements
Vec(13)     for element 13-th
Vec(12:)    for elements 12-th to last
Vec(:10)    for elements first to 10-th
Vec(5:8)    for elements 5-th to 8-th

Vec(3,100) (2-dimensional with 3 columns by 100 rows):
```

may be addressed by:

```
Vec(2,5:8)   for elements 5-th to 8-th in 2-nd column
Vec(2:3,5:8) for elements 5-th to 8-th in 2-nd to 3-rd columns
Vec(2,5)     for element 5-th in 2-nd column
Vec(:,3)     for all elements in 3-rd row
Vec(2)       for all elements in 2-nd column (SPECIAL CASE)
```

The latest line shows the special (and non-standard with FORTRAN 77) notation such that missing indexes are substituted to the right.

An 'invisible' vector called '?', mono-dimensional and of length 100, is always present. Is is used for communicating between user arrays and KUIP vectors, being equivalenced with the real array VECTOR(100) in the labeled common block /KCWORK/.

VECTOR/CREATE vname [type values]

VNAME	C	"Vector name(length)"
TYPE	C	"Vector type" D='R'
VALUES	C	"Value list" D='␣' Separate Vararg

Possible TYPE values are:

R
I

Create a vector named VNAME (elements are set to zero). The dimensions are taken from the name, for example VEC(20), VEC(3,100), VEC(2,2,10). Up to 3 dimensions are supported. Dimensions which are not specified are taken to 1, for example VEC(10) —⟩ VEC(10,1,1) and VEC —⟩ VEC(1,1,1). The vector may be of type Real or Integer. A vector is filled at the same time if parameters are given after the TYPE:

```
VEC/CREATE V(10) R 1 2 3 4 5 66 77 88 99 111
VEC/CREATE W(20) R 1 2 3
```

In the last example only the first three elements are filled. Vector elements may be changed later with the command VECTOR/INPUT.

If many equal values have to be entered consecutively, one can specify just one value and precede it by a repetition factor and an asterisk. Example:

```
VEC/CREATE Z(20) R 5*1 2 4*3   --->   VEC/CREATE Z(20) R 1 1 1 1 1 2 3 3 3
3
```

Enter HELP VECTOR for more information on vector addressing.

VECTOR/LIST

List all vectors (name, dimensions, type).

VECTOR/DELETE vlist

VLIST C "Vector list" D=' ' Loop

Delete from memory all vectors in the list VLIST. The vectors are separated in the list by a comma and embedded blanks are not allowed. An asterisk at the end of VLIST acts as wild-card:

```
VEC/DEL AB*          --->  deletes all vectors starting by AB
VEC/DEL *            --->  deletes all vectors
```

VECTOR/COPY vnam1 vnam2

VNAM1 C "Source vector name"

VNAM2 C "Destination vector name"

Copy a vector into another one. Mixed vector type copy is supported (e.g. Integer —⟩ Real and viceversa). If VNAM2 does not exist it is created with the required dimensions, not necessarily the same as the source vector if a sub-range was specified. For example, if A is a 3 x 100 vector and B does not exist, COPY A(2,11:60) B will create B as a 50 elements mono-dimensional vector; a special (and non-standard with FORTRAN 77) notation is used such that, still using the above vectors, COPY A(2,1:100) B and COPY A(2) B have the same effect.

Note that VECTOR/COPY does not allow a range for the destination vector not specifying consecutive elements (i.e. along the first dimension):

```
VEC/COPY V(5)      W(3,4)     | O.K.
VEC/COPY V1(2:3,5) V2(4:5,9)  | O.K.
VEC/COPY V1(5,2:3) V2(4:5,9)  | O.K.
VEC/COPY V1(3,3:4) V2(4,4:5)  | NOT allowed
VEC/COPY V1(2:3,5) V2(2,4:5)  | NOT allowed
```

Enter HELP VECTOR for more information on vector addressing.

VECTOR/INPUT vname [values]

VNAME	C	"Vector name"
VALUES	C	"Value list" D='␣' Separate Vararg

Enter values into a vector from the terminal. Example:

```
VEC/INPUT V(6:10) 1.1 2.22 3.333 4.4444 5.55555
```

If many equal values have to be entered consecutively, one can specify just one value and precede it by a repetition factor and an asterisk. Example:

```
VEC/INPUT V 5*1 2 4*3   --->   VEC/INPUT V 1 1 1 1 1 2 3 3 3 3
```

Enter HELP VECTOR for more information on vector addressing.

VECTOR/PRINT vname [dense]

VNAME	C	"Vector name"
DENSE	I	"Output density" D=1 R=0,1,2

Write to the terminal the content of a vector. Enter HELP VECTOR for more information on vector addressing.

If DENSE.EQ.0 the output is one vector element per line. If DENSE.EQ.1 the output for a sequence of identical vector elements is compressed to two lines stating the start and end indices. If DENSE.EQ.2 the output for a sequence of identical vector elements is compressed to a single line.

VECTOR/READ vlist fname [format opt match]

VLIST	C	"Vector list"
FNAME	C	"File name" D='␣'
FORMAT	C	"Format" D='␣'
OPT	C	"Options" D='OC'
MATCH	C	"Matching pattern" D='␣'

Possible `OPT` values are:

`OC`
`O`
`'␣'`
`C`

Enter values into vector(s) from a file. A format can be specified, e.g. FORMAT='F10.5,2X,F10.5', or the free format is used if FORMAT is not supplied.

If vector(s) are not existing they will be created of the size as read from the file.

Vectors in the list VLIST are separated by a comma and embedded blanks are not allowed. If subscripts are present in vector names, the smallest one is taken.

OPT is used to select between the following options:

```
'OC'   file is Opened, read and then Closed (default case)
'O'    file is Opened and then read (left open for further reading)
' '    file is read (already open, left so for further reading)
'C'    file is read and then Closed (already open)
```

If the character 'Z' is present in OPT, the vector elements equal to zero after reading are set to the latest non-zero element value (for example reading 1 2 3 0 0 4 0 5 will give 1 2 3 3 3 4 4 5).

MATCH is used to specify a pattern string, restricting the vector filling only to the records in the file which verify the pattern. Example of patterns:

```
 /string/      match a string (starting in column 1)
-/string/      do not match a string (starting in column 1)
 /string/(n)   match a string, starting in column n
 /string/(*)   match a string, starting at any column
```

Enter HELP VECTOR for more information on vector addressing.

VECTOR/WRITE vlist [fname format chopt]

VLIST	C	"Vector list"
FNAME	C	"File name" D='␣'
FORMAT	C	"Format" D='5(1X,G13.7)'
CHOPT	C	"Options" D='OC'

Possible CHOPT values are:

OC

O

'␣'

C

Write to a file the content of vector(s). If FNAME=' ' the content is written to the terminal. A format can be specified, e.g. FORMAT='F10.5,2X,F10.5', or the default one is used if FORMAT is not supplied.

Vectors in the list VLIST are separated by a comma and embedded blanks are not allowed. If subscripts are present in vector names, the smallest one is taken.

CHOPT is used to select between the following options:

```
'OC'   file is Opened, written and then Closed (default case)
'O'    file is Opened and then written (left open for further writing)
' '    file is written (already open, left so for further writing)
'C'    file is written and then Closed (already open)
```

Enter HELP VECTOR for more information on vector addressing.

5.13 Menu VECTOR/OPERATIONS

Simple arithmetic operations between vectors. In all the operations only the minimum vector length is considered, i.e. an operation between a vector A of dimension 10 and a vector B of dimension 5 will involve the first 5 elements in both vectors. If the destination vector does not exist, it is created with the same length as the source vector.

`VECTOR/OPERATIONS/VBIAS vnam1 bias vnam2`

`VNAM1`	C	"Source vector name"
`BIAS`	R	"Bias value"
`VNAM2`	C	"Destination vector name"

VNAM2(I) = BIAS + VNAM1(I)

`VECTOR/OPERATIONS/VSCALE vnam1 scale vnam2`

`VNAM1`	C	"Source vector name"
`SCALE`	R	"Scale factor"
`VNAM2`	C	"Destination vector name"

VNAM2(I) = SCALE * VNAM1(I)

`VECTOR/OPERATIONS/VADD vnam1 vnam2 vnam3`

`VNAM1`	C	"First source vector name"
`VNAM2`	C	"Second source vector name"
`VNAM3`	C	"Destination vector name"

VNAM3(I) = VNAM1(I) + VNAM2(I)

`VECTOR/OPERATIONS/VMULTIPLY vnam1 vnam2 vnam3`

`VNAM1`	C	"First source vector name"
`VNAM2`	C	"Second source vector name"
`VNAM3`	C	"Destination vector name"

VNAM3(I) = VNAM1(I) * VNAM2(I)

`VECTOR/OPERATIONS/VSUBTRACT vnam1 vnam2 vnam3`

`VNAM1`	C	"First source vector name"
`VNAM2`	C	"Second source vector name"
`VNAM3`	C	"Destination vector name"

VNAM3(I) = VNAM1(I) - VNAM2(I)

`VECTOR/OPERATIONS/VDIVIDE vnam1 vnam2 vnam3`

`VNAM1`	C	"First source vector name"
`VNAM2`	C	"Second source vector name"
`VNAM3`	C	"Destination vector name"

VNAM3(I) = VNAM1(I) / VNAM2(I) (or 0 if VNAM2(I)=0)

A KUIP Programming example

As an example of how to implement a user interface with KUIP the code for a simple Reverse Polish Notation pocket calculator is presented.

A.1 The Command Definition File

The CDF for the RPN calculator

```
>Name RPNDEF

>Menu RPN
>Guidance
Reverse Polish Notation sub-pocket calculator
using KUIP for the user interface.

>Command ENTER
>Parameters
+
NUM 'Enter a number' R D=0.
>Guidance
Push the number(s) given as parameter(s) into the stack.
If none push a zero.
>Action RPENT

>Command ADD
>Guidance
Push the number(s) given as parameter(s) into the stack.
Add the two upper-most numbers of the stack and shift it up.
>Action RPOPER

>Command SUBTRACT
>Guidance
Push the number(s) given as parameter(s) into the stack.
Subtract the two upper-most numbers of the stack and shift it up.
>Action RPOPER

>Command MULTIPLY
>Guidance
Push the number(s) given as parameter(s) into the stack.
Multiply the two upper-most numbers of the stack and shift it up.
>Action RPOPER
```

```
>Command DIVIDE
>Guidance
Push the number(s) given as parameter(s) into the stack.
Divide the two upper-most numbers of the stack and shift it up.
>Action RPOPER

>Command PRINT
>Guidance
Print the content of the stack.
>Action RPRINT

>Command CLEAR
>Guidance
Clear the stack.
>Action RPCLR
```

A.2 The application program

FORTRAN code for the RPN calculator application program

```
      PROGRAM RPN
      COMMON/PAWC/PAW(50000)
      COMMON/RPNSTK/ISTACK,STACK(100)
*---> Initialize ZEBRA and the store /PAWC/
      CALL MZEBRA(-3)
      CALL MZPAW(50000,' ')
*---> Initialize KUIP with 5000 words as minimum division size
      CALL KUINIT(5000)
*---> Create user command structure from definition file (command
*---> definition routine name RPNDEF defined in CDF
*---> with '>Name RPNDEF')
      CALL RPNDEF
*---> Change the prompt by executing the command SET/PROMPT
      CALL KUEXEC('SET/PROMPT ''RPN >''')
*---> Initialize the stack in /RPNSTK/
      CALL RPCLR
*---> Give control to KUIP (with no 'STYLE G', call KUWHAG
*---> to get 'STYLE G')
      CALL KUWHAT
*---> Typing 'QUIT' or 'EXIT' we return here
      END

      SUBROUTINE RPCLR
      COMMON/RPNSTK/ISTACK,STACK(100)
      DO 10 I=1,100
        STACK(I)=0.
10    CONTINUE
      ISTACK=50
      CALL RPRINT
      END
```

```
      SUBROUTINE RPENT
      COMMON/RPNSTK/ISTACK,STACK(100)
      CHARACTER*32 CMD
      CALL KUPATL(CMD,NPAR)
      IF(CMD.EQ.'ENTER' .AND. NPAR.EQ.0) NPAR=1
      DO 10 I=1,NPAR
        CALL KUGETR(R)
        ISTACK=ISTACK+1
        IF(ISTACK.GT.100) THEN
          PRINT *,'ERROR: Stack overflow'
          GOTO 20
        ENDIF
        STACK(ISTACK)=R
10    CONTINUE
20    CONTINUE
      IF(CMD.EQ.'ENTER') CALL RPRINT
      END

      SUBROUTINE RPOPER
      COMMON/RPNSTK/ISTACK,STACK(100)
      CHARACTER*32 CMD
      CALL KUPATL(CMD,NPAR)
      CALL RPENT
      IF(ISTACK.LT.2) THEN
        PRINT *,'ERROR: Stack underflow'
        GOTO 10
      ENDIF
      IF(CMD.EQ.'ADD') THEN
        STACK(ISTACK-1)=STACK(ISTACK)+STACK(ISTACK-1)
      ELSEIF(CMD.EQ.'SUBTRACT') THEN
        STACK(ISTACK-1)=STACK(ISTACK)-STACK(ISTACK-1)
      ELSEIF(CMD.EQ.'MULTIPLY') THEN
        STACK(ISTACK-1)=STACK(ISTACK)*STACK(ISTACK-1)
      ELSEIF(CMD.EQ.'DIVIDE') THEN
        IF(STACK(ISTACK).EQ.0) THEN
          PRINT *,'ERROR: Divide by zero'
        ELSE
          STACK(ISTACK-1)=STACK(ISTACK-1)/STACK(ISTACK)
        ENDIF
      ENDIF
      ISTACK=ISTACK-1
10    CONTINUE
      CALL RPRINT
      END

      SUBROUTINE RPRINT
      COMMON/RPNSTK/ISTACK,STACK(100)
      PRINT *,(STACK(I),I=ISTACK,ISTACK-3,-1)
      IF(STACK(ISTACK).GT.0) THEN
        PRINT *,'********'
      ELSE
```

```
          PRINT *,'*********'
        ENDIF
        END
```

A.3 Example of a run

Example of a session using the RPN calculator

```
$ rpn
 0.0000000 0.0000000 0.0000000 0.0000000
 *********
 RPN > style an

 From  /...

  1:   KUIP          Command Processor commands.
  2:   MACRO         Macro Processor commands.
  3:   VECTOR        Vector Processor commands.
  4:   RPN           Reverse Polish Notation sub-pocket calculator

 Enter a number ('Q'=command mode): 4

 From  /RPN/...

  1: * ENTER         Push the number(s) given as parameter(s) into the stack.
  2: * ADD           Push the number(s) given as parameter(s) into the stack.
  3: * SUBTRACT      Push the number(s) given as parameter(s) into the stack.
  4: * MULTIPLY      Push the number(s) given as parameter(s) into the stack.
  5: * DIVIDE        Push the number(s) given as parameter(s) into the stack.
  6: * PRINT         Print the content of the stack.
  7: * CLEAR         Clear the stack.

 Enter a number ('\'=one level back, 'Q'=command mode): 1

  * /RPN/ENTER  [ NUM ]

 Add parameters or just <CR> to the command line
 ('#'=cancel execution, '?'=help) :

 /RPN/ENTER 1 2 3
 3.000000 2.000000 1.000000 0.0000000
 ********

 From  /RPN/...

  1: * ENTER         Push the number(s) given as parameter(s) into the stack.
  2: * ADD           Push the number(s) given as parameter(s) into the stack.
  3: * SUBTRACT      Push the number(s) given as parameter(s) into the stack.
  4: * MULTIPLY      Push the number(s) given as parameter(s) into the stack.
```

```
 5: * DIVIDE       Push the number(s) given as parameter(s) into the stack.
 6: * PRINT        Print the content of the stack.
 7: * CLEAR        Clear the stack.

Enter a number ('=one level back, 'Q'=command mode): 2

 * /RPN/ADD

Add parameters or just <CR> to the command line
('#'=cancel execution, '?'=help) :

/RPN/ADD
5.000000 1.000000 0.0000000 0.0000000
********
From  /RPN/...

 1: * ENTER        Push the number(s) given as parameter(s) into the stack.
 2: * ADD          Push the number(s) given as parameter(s) into the stack.
 3: * SUBTRACT     Push the number(s) given as parameter(s) into the stack.
 4: * MULTIPLY     Push the number(s) given as parameter(s) into the stack.
 5: * DIVIDE       Push the number(s) given as parameter(s) into the stack.
 6: * PRINT        Print the content of the stack.
 7: * CLEAR        Clear the stack.

Enter a number ('=one level back, 'Q'=command mode): q

RPN > print
5.000000 1.000000 0.0000000 0.0000000
********
RPN > enter 1 2 3
3.000000 2.000000 1.000000 5.000000
********
RPN > add
5.000000 1.000000 5.000000 1.000000
********
RPN > add
6.000000 5.000000 1.000000 0.0000000
********
RPN > add
11.00000 1.000000 0.0000000 0.0000000
********
RPN > add 5
16.00000 1.000000 0.0000000 0.0000000
********
RPN > add 10 20
30.00000 16.00000 1.000000 0.0000000
********
RPN > usage rpn/

 * RPN/ENTER  [ NUM ]
 * RPN/ADD
 * RPN/SUBTRACT
 * RPN/MULTIPLY
```

```
  * RPN/DIVIDE
  * RPN/PRINT
  * RPN/CLEAR

 RPN > help clear

  * /RPN/CLEAR

    Clear the stack.

 RPN > c
 *** Ambiguous command. Possible commands are :

 /KUIP/ALIAS/CREATE
 /KUIP/SET_SHOW/COMMAND
 /KUIP/SET_SHOW/COLUMNS
 /MACRO/SYNTAX/Branching/CASE
 /VECTOR/CREATE
 /VECTOR/COPY
 /RPN/CLEAR

 RPN > r/c
 0.0000000 0.0000000 0.0000000 0.0000000
 *********
 RPN > help rpn/print

  * /RPN/PRINT

    Print the content of the stack.

 RPN > help en

  * /RPN/ENTER  [ NUM ]

    NUM        R 'Enter a number' D=0

    Push the number(s) given as parameter(s) into the stack.
    If none push a zero.

 RPN > q
$
```

B Interfacing Fortran and C

Over the past two years the originally Fortran-based KUIP was gradually rewritten in C. The rewrite was not a pure “exercise of style” but it appeared to be the best choice in order to improve the quality of the package. Key elements for a user interface package such as dynamic string handling, recursivity, and function pointers are inherent to the C language. Using C allowed to remove most of the limitations with respect to maximum string lengths built into the original Fortran Code. It also facilitated the interface to Motif which is only accessible through C libraries.

The Fortran routines of KUIP have been replaced by C functions. The C code emulates the former Fortran entry points and is therefore completely compatibility.

When doing inter-language calls between Fortran and C we have to realize that the Fortran part is fixed, and that the C code has to adapt to the characteristics employed by the Fortran compiler. The `cfortran.h`[1] header file by Burkhard Burow provides convenient preprocessor macros for interfacing Fortran and C. For reasons explained further down it does, however, not cover the IBM mainframe systems. Also, `cfortran.h` exercises the highest degree of `cpp` magic and always which involves the risk of stretching the C preprocessor beyond its limits.

By no means we want to discourage the use of `cfortran.h`. However, if `cfortran.h` cannot be used (because the target platform is not supported or the overhead may be considered too large) here you should find the necessary information for doing the Fortran-C interface “by hand”.

B.1 Inter-language Calls

B.1.1 Type Definitions

It is good style to use `typedef` names to map the Fortran data types to their C counterparts. Up to now all platforms are satisfied by a single set of definitions:

```
typedef int     INTEGER;
typedef INTEGER LOGICAL;
typedef float   REAL;
typedef double  DBLPREC;
typedef struct { REAL re; REAL im; } COMPLEX;
```

but, for example, a Fortran compiler with `AUTODBL` option for generation 64-bit code would require different mappings.

[1] `cfortran.h` and `f2h` for generating interface definitions from Fortran source code are available as part of the CERN program library distribution.

B.1.2 External Identifiers Naming

First of all, there are differences in the naming of external identifiers (SUBROUTINE, FUNCTION and COMMON block names in Fortran and their C equivalent).

B.1.2.1 SUBROUTINE

For a Fortran CALL SUB the corresponding C routine has to be named:

SUB	all uppercase on Cray with cft77 compiler case-insensitive on IBM/370 and VMS
sub	all lowercase on Apollo with ftn compiler
sub_	lowercase with underscore added on all other system

The HP/UX and RS-6000 Fortran compilers need a special switch (+ppu and -qextname, respectively) in order to generate sub_ instead of sub. CERN program library policy is to use the form with the appended underscore to avoid the risk of name clashes between Fortran user routines and C library functions.

For convenience we define the capitalized form Sub as a preprocessor identifier:

```
#define Sub F77_NAME(sub,SUB)
```

where the preprocessor macro F77_NAME maps to the appropriate external name:

```
/* Cray, VM/CMS, MVS, VMS */
#define F77_NAME(name,NAME) NAME

/* Apollo FTN */
#define F77_NAME(name,NAME) name

/* others */
#define F77_NAME(name,NAME) name##_
```

The IBM C/370 compiler requires in addition a

```
#pragma linkage(SUB,FORTRAN)
```

in order to use the correct argument passing mechanism which is different between Fortran and C. Unfortunately the #pragma cannot be included into the F77_NAME macro and must be specified separately.

B.1.2.2 FUNCTION

The same rules as for SUBROUTINEs apply also for FUNCTION names. We found, however, that only INTEGER and LOGICAL FUNCTIONs can be emulated in C. REAL FUNCTIONs and others are not guaranteed to work on all platforms.

B.1.2.3 COMMON

Fortran `COMMON` blocks can be accessed from C code. Most compilers apply the same rule for generating the external identifier as for routines but up to now we came across two exceptions: The Convex compiler adds an underscore both in front and at the end. The Absoft Fortran compiler for the NeXT uses the plain lowercase name without underscores.

Again for convenience we define the capitalized form `Com` as a preprocessor identifier

```
#define Com F77_BLOCK(com,COM)
```

where the preprocessor macro expands to the appropriate identifier:

```
/* Convex */
#define F77_BLOCK(name,NAME)  _##name##_

/* NeXT */
#define F77_BLOCK(name,NAME)  name

/* others */
#define F77_BLOCK(name,NAME)  F77_NAME(name,NAME)
```

Naively one would assume that the C structure mapping the `COMMON` block should be a *declaration* (i.e. "`extern struct ...`") since the storage should be allocated by the Fortran side. Some platforms, however, require the form of a *definition*, i.e. leaving out the `extern` keyword. In order to cope with this difference we define

```
/* Apollo, Convex, Cray */
#define EXTERN

/* others */
#define EXTERN extern
```

One last complication is that on Apollo the C definition has to be given a special attribute. Otherwise the linker allocates the C structure separate from the `COMMON` block it should map. Again we can define a system dependent preprocessor macro to cope with this difference:

```
/* Apollo */
#define F77_COMMON(name) name __attribute((__section(name)))

/* others */
#define F77_COMMON(name) name
```

Having these ingredients it is straightforward to translate any Fortran `COMMON` block into a C `struct`. For example,

```
      INTEGER I
      REAL X
      DOUBLE PRECISION D
      COMMON /COMX/ I,X(3,3),D
```

```
      CHARACTER*80 CHTEXT(10)
      COMMON /COMC/ CHTEXT
```

is described by

```
#define Comx F77_BLOCK(comx,COMX)

EXTERN struct {
  INTEGER i;
  REAL x[3][3];
  DBLPREC d;
} F77_COMMON(Comx);

#define Comc F77_BLOCK(comc,COMC)

EXTERN struct {
  char text[10][80];
} F77_COMMON(Comc);
```

and the values are accessible as, for example

```
Comx.i
Comx.x[1][2]     for  X(3,2)
Comc.text[4][0]  for  CHTEXT(5)(1:1)
```

Note that one should not rely on the possibility to extend the `COMMON` block size by changing the dimensions in the C code. It is platform dependent whether

```
EXTERN struct {
  char text[1000][80];
} F77_COMMON(Comc);
```

without changing the Fortran definition accordingly will actually reserve the requested amount of memory for the `COMMON` block.

Also one has to be careful with the placement of `DOUBLE PRECISION` variables. They should always be at an even offset from the start of the `COMMON` block because otherwise in a case like

```
      REAL X,Y
      DOUBLE PRECISION D
      COMMON /COM/ X,D,Y
```

it is not guaranteed that it can be mapped as

```
struct {
  REAL x;
  DBLPREC d;
  REAL y;
} ...
```

While the C compiler is allowed to add padding in order to align `d`, the Fortran compiler is obliged to allocate `X` and `D` at consecutive machine words. Many CPUs signal a bus error upon unaligned access to `DOUBLE PRECISION` variables, and compilers usually add protective code and issue a warning message.

One exception is the HP/UX Fortran compiler which silently adds padding in the same way the C compiler does. That makes the Fortran-C interface easier but at the same time violates the Fortran storage association. In principle, the same `COMMON` block could be declared in a different routine as

```
      REAL Z
      COMMON /COM/ Z(4)
```

and for the HP/UX Fortran compiler `Y` and `Z(4)` do not refer to the same memory location anymore.

B.1.2.4 /PAWC/

Some of the complications in accessing `COMMON` blocks from C could be avoided by defining a structure pointer which is filled by a call from a Fortran routine passing the first variable in the `COMMON` block.

In fact in KUIP this possibility is used for the `/PAWC/` Zebra store. The C structure definition looks like

```
#define Pawc kc_pawc

EXTERN struct COMMON_PAWC {
  INTEGER NWPAR;
  INTEGER IXPAWC;
  INTEGER IHBOOK;
  INTEGER IXHIGZ;
  INTEGER IXKUIP;
  INTEGER IFENCE[5];
  INTEGER LQDATA[8];
  INTEGER IQDATA[999];
} *Pawc;
```

and the pointer is initialized by a `CALL KIPAWC(NWPAR)` to

```
void Kipawc( struct COMMON_PAWC *pawc )
{
  Pawc = pawc;
}
```

Linking to `/PAWC/` in this way was introduced due to a request from W. Wojcik of IN2P3 in Lyon. They wanted to exploit the option `DC(name)` of the VS-Fortran compiler which allows to allocate `COMMON` blocks at execution time. The generated code references `COMMON` block variables indirectly through a pointer to dynamic memory rather than as an absolute address resolved by the loader. (The libraries produced on CERNVM do not make use of the `DC` option. The C code works either way without change.)

Preprocessor macros allow to write C code accessing the Zebra store in a Fortran-ish style. For example, the equivalent of

```
      DO 10 I = 1, IQ(LX-1)
        Q(LX+1) = 3.14
 10   CONTINUE
```

can be written as

```
#define LQ(n)           Pawc->LQDATA[n-1]
#define IQ(n)           Pawc->IQDATA[n-1]
#define  Q(n) (((REAL*)(Pawc->IQDATA))[n-1])

  for( i = 1; i <= IQ(lx-1); i++ ) {
    Q(lx+1) = 3.14;
  }
```

B.1.3 Calling C from Fortran

B.1.3.1 Numeric data

Passing numeric arguments between Fortran and C is straightforward. One only has to remember that Fortran always uses *call-by-reference*, i.e. it passes the address of the memory location where the value is stored. Consequently, the Fortran routine call

```
      CALL SUB(X,2)
```

corresponds to the C function declaration:

```
void Sub( REAL* xp, INTEGER* ip )
```

The value itself is accessed as in

```
float x = *xp;
```

To store a value back into the Fortran variable `X` one has to write in C

```
*xp = 3.14;
```

B.1.3.2 Arrays

If `X` is an array the third element `X(3)` (in Fortran) has to be addressed as `xp[2]`, since in C array indices start at 0.

It is important to note that the pointers received as arguments **must not** be used as local variable. For example, one might be tempted to write:

```
void Sub( REAL* xp, INTEGER* ip )
{
  int i;
  for( i = *ip; i > 0; i-- ) {
    float x = *xp++;
    ...
  }
}
```

where `xp` is incremented to point to the next element, which is perfectly legal C. But there is a potential problem if `Sub` is called from Fortran.

Some machines (e.g. IBM/370 and Convex) use argument blocks, i.e. the call passes the address of a memory region containing the argument addresses. Since a Fortran routine `SUB` would have no way the alter the content of the argument block, the Fortran compiler treats it as a compile time constant. As a result, only the first call to `Sub` will access the correct array elements.

B.1.3.3 LOGICAL

The representation of `.TRUE.` varies between platforms. It seems to be safe to test a `LOGICAL` input against `0` but output parameters should use the proper `.TRUE.` and `.FALSE.` constants. Returning the C values `~0` or `0` will work in tests such as

```
IF(L) ...
```

but may fail for

```
IF(L .EQ. .TRUE.) ...
```

B.1.3.4 EXTERNAL

C entry points can receive Fortran `EXTERNAL` arguments as in

```
EXTERNAL XT
CALL SUB(XT)
```

Most platforms pass directly the address of the routine

```
typedef void SUBROUTINE();

void Sub( SUBROUTINE* xt )
```

but some compilers (AIX/370, IBM/370, Apollo, NeXT) store the routine address in a memory cell whose address the called function receives. If we undo the indirection

```
void Sub( SUBROUTINE** xtp )
{
  SUBROUTINE* xt = *xtp;
  ...
}
```

in the routine header we can use afterwards in both cases the same C statement

```
(*xt)(...)
```

for calling the routine through the function pointer.

Using a `typedef` name does more than just improve the readability of the declaration. For the IBM C/370 compiler it is essential to add

```
#pragma linkage(SUBROUTINE,FORTRAN)
```

in order to tell that function pointers of type `SUBROUTINE` have to use the Fortran convention of argument passing.

B.1.3.5 CHARACTER

This is one of the most tricky and tedious part of Fortran/C intermixing. A Fortran `CHARACTER` variable is characterized by two informations, the address and the length, and the mechanism for passing these two values is compiler dependent. We can label the different mechanisms according to the number of arguments received by the called routine. For a `SUBROUTINE` with `N` parameters in total of which `k` are of type `CHARACTER` the following number of arguments can be passed:

`N` Address and length are passed in a single string descriptor argument.

`N+k` The address is passed at the position of the `CHARACTER` argument and for each `CHARACTER` argument the length information is added at the end of the regular argument list.

`2N` Like the previous case but there is an extra length indicator for each `CALL` argument.

With variations on the type of the length indicator these are the three different schemes I have come across up to now. Of course, there are more possible permutations a new Fortran compiler may choose from.

To give a more complete picture we will take the example of the following basic Fortran routine

```
      SUBROUTINE SUB(X,CH,Y)
      REAL X,Y
      CHARACTER*(*) CH
```

and give the different headers, according to machines, for the C equivalent function in order to make it Fortran-callable without any problem. In each case we will end up in the C function with the fixed variable names `char* ch_ptr` and `int ch_len` containing the address and length of `CH`.

<u>`VMS`</u>

VMS passes the address of a string descriptor containing the address and length information as structure elements:

```
#include <descrip.h>

void Sub( REAL* x,
          struct dsc$descriptor_s* ch_dsc,
          REAL* y )
{
  char* ch_ptr = ch_dsc->dsc$a_pointer;
  int   ch_len = ch_dsc->dsc$w_length;
  ...
}
```

<u>`Cray`</u>

Cray packs both informations into the 64-bit word received for each argument (and the bit fields used are actually processor dependent):

```
#include <fortran.h>

void Sub( REAL* x, _fcd ch_dsc, REAL* y )
{
  char* ch_ptr = _fcdtocp(ch_dsc);
  int   ch_len = _fcdlen(ch_dsc);
```

```
  ...
}
```

VS-Fortran

IBM VS-Fortran uses the `2N` scheme with *call-by-reference* for the length information:

```
#pragma linkage(SUB,FORTRAN)

void Sub( REAL* x,     char* ch_ptr, REAL* y,
          int*  x_size, int* ch_ref, int*  y_size )
{
  int ch_len = *ch_ref;
  ...
}
```

NeXT

For the Absoft Fortran compiler on NeXT it looks similar except that *call-by-value* is used:

```
void Sub( REAL* x,    char* ch_ptr, REAL* y,
          int   x_size, int ch_len, int   y_size )
{
  ...
}
```

Others

The bulk of Fortran compilers adds the length as `int` only for type `CHARACTER`:

```
void Sub( REAL* x, char* ch_ptr, REAL* y,
                     int ch_len )
{
  ...
}
```

Apollo

The only remaining exception is the Apollo `ftn` compiler which passes the length in a `short*`:

```
void Sub( REAL* x, char* ch_ptr, REAL* y,
                  short* ch_ref )
{
  int ch_len = *ch_ref;
  ...
}
```

KUIP offers a number of convenience functions (`fstrdup`, `fstrset`, ...) to convert between fixed length, blank-filled Fortran `CHARACTER` buffers and null-terminated, blank-trimmed C strings. For details refer to chapter 4.

B.1.4 Calling Fortran from C

Calling a Fortran routine from C is basically the reverse operation of providing a C routine which is Fortran-callable. One has to know the correct spelling for the external identifier, for numeric variables pass the address instead of the value itself, and for `CHARACTER` items construct the appropriate argument list.

We continue to use the example of a subroutine with the same calling sequence `SUB(X,CH,Y)`. In the following code fragments we assume that

```
REAL  x, y;
char* ch_ptr;
int   ch_len;
```

are already filled and that we want to pass their values to Fortran. Again the platform dependence in passing character data causes the main difficulty.

<u>VMS</u>

On VMS we have to fill the elements of the string descriptor and then pass its address:

```
#include <descrip.h>

  struct dsc$descriptor_s ch_dsc;
  ch_dsc.dsc$b_dtype   = DSC$K_DTYPE_T;
  ch_dsc.dsc$b_class   = DSC$K_CLASS_S;
  ch_dsc.dsc$w_length  = ch_len;
  ch_dsc.dsc$a_pointer = ch_ptr;
  Sub( &x, &ch_dsc, &y );
```

<u>Cray</u>

On Cray a predefined macro packs the address and length information into a `cft77` descriptor:

```
#include <fortran.h>

  Sub( &x, _cptofcd(ch_ptr,ch_len), &y );
```

<u>NeXT</u>

```
  Sub(        &x,  ch_ptr,        &y,
       (sizeof x), ch_len, (sizeof y) );
```

<u>Apollo ftn</u>

The byte ordering of the Motorola-68k CPU does not allow to pass the address of a 4-byte `int` if a 2-byte `short` is expected. Therefore for the Apollo `ftn` compiler we have to use an auxiliary variable:

```
short ch_ref = ch_len;
Sub( &x, ch_ptr, &y,
        &ch_ref );
```

Others

For all other compilers except IBM-C/370 the call is straightforward:

```
Sub( &x, ch_ptr, &y,
         ch_len );
```

C/370

The IBM C/370 compiler **does not** allow to call a VS-Fortran routine with CHARACTER arguments. The naive approach

```
#pragma linkage(SUB,FORTRAN)

  int x_size = (sizeof x);
  int y_size = (sizeof y);
  Sub( &x,       ch_ptr, &y,
       &x_size, &ch_len, &y_size );
```

does not work. VS-Fortran passes the argument count implicitly by setting the sign bit of the last word in the argument block. uses argument blocks containing the reference addresses.

For routines with CHARACTER arguments both the words at positions N and 2N must be flagged. The #pragma linkage instructs the C compiler to create a Fortran-style argument block with the last word flagged. Unfortunately there is no way to set the sign bit also in the middle of the argument block. Attempts to patch the argument list such as

```
REAL* y_ref =
  (REAL*)(0x80000000 | (int)&y);
Sub( &x,       ch_ptr,  y_ref,
     &x_size, &ch_len, &y_size );
```

do not succeed. (It seems that C/370 clears the sign bit again before storing the address into the argument block.)

The work-around is to use a wrapper routine which receives the character arguments as Hollerith data and copies them into temporary CHARACTER variables. For example,

```
#pragma linkage(K77XCX,FORTRAN)
#pragma linkage(SUB,FORTRAN)
extern SUBROUTINE SUB;

  SUBROUTINE* sub_ptr = SUB;
  K77XCX( &sub_ptr, &x, ch_ptr, &ch_len, &y );
```

with

```
      SUBROUTINE K77XCX(SUB,X1,K2,L2,X3)
      EXTERNAL SUB
      PARAMETER(MBIG=256)
      CHARACTER*(MBIG) C2
      CALL UHTOC(K2,4,C2,L2)
      CALL SUB(X1,C2(:L2),X3)
      CALL FMEMCPY(K2,C2(:L2))
      END
```

Obviously, `MBIG` should be set to the maximum string length ever expected. `FMEMCPY` is a Fortran-callable version of `memcpy` provided by KUIP. The `CHARACTER` data has to be copied back into the C string in case that it is an output parameter. (`UCTOH` cannot be used because there the number of bytes copied is rounded up to the next multiple of 4.)

The disadvantage is that each type of argument list requires its own wrapper. This is not foreseen in the `cfortran.h` approach and is the reason why it cannot be used for C/370.

B.2 Input/Output

With few exceptions mixing Fortran and C input/output calls is asking for trouble. Often the runtime libraries of both languages implement separate buffering schemes in order to improve performance.

Terminal Input

One should never mix `READ(5,...)` and `fgets(...,stdin)` requests. On most systems it works as long as the input device is really a terminal. However, at the latest when `stdin` is connected to a file or a pipe the Fortran `READ` will suck in more than just its share to satisfy the immediate request. Any following `fgets` will then access the wrong information (or report end-of-file).

Terminal Output

Mixing `WRITE(6,...)` and `fprintf(stdout,...)` is save as long as the output device is the terminal. When `stdout` is redirected the C stream functions usually start buffering the output.

On a number of Unix systems (RS/6000, Alpha/OSF, Ultrix) the Fortran runtime library does not use the C stream functions, and the Fortran and C output buffers be flushed out of sequence. The C runtime libraries on these systems can be told be limit themselves to line buffering by

```
if( !isatty( fileno(stdout) ) )
  setlinebuf( stdout );
```

It seems that the Fortran runtime libraries are already disciplined enough to use for `WRITE(6,...)` and `PRINT~...` only line buffering. Note that `setlinebuf` is not a standard Unix function. Hopefully if it is not contained in `stdio.h` it also is not necessary.

Formatted I/O

There is no general way to associate a Fortran logical unit number with a C `FILE*` stream. KUIP offers C functions `ku_read` and `ku_write` as a convenient interface to Fortran I/O statements for reading/writing complete lines. For details refer to chapter 4.

(Note that even in Fortran a `READ(LUN,'(A)') CHLINE` is not portable because the VS-Fortran runtime library is not able to handle V-format files.)

Unformatted I/O

Some systems provide a function `getfd` to inquire the C file descriptor associated to a Fortran logical unit number. This is however non-standard, and often Fortran unformatted sequential I/O adds system dependent control information to allow for variable length records.

The CIO package in KERNLIB provides a Fortran-callable set of interface routines to the C `read` and `write` functions in order to perform system independent unformatted I/O, both for sequential and direct access with fixed record sizes.

B.3 Initialization

On many systems the runtime libraries for a given language has to be initialized — a task which is then performed by the main program.

B.3.1 Fortran main program

With a Fortran main program calling C functions directly or indirectly the following points have to be respected.

C/370

The C/370 runtime library is automatically initialized when the first C routine is called. The system resources allocated there are released again as soon as the Fortran routine, from which the C routine triggering the initialization was called, returns. Calling another C routine after that does not cause a re-initialization and the program usually crashes because the `malloc` heap has become invalid.

This effect can be safely avoided by calling the first C routine immediately in the Fortran main program. KERNLIB contains a dummy routine `INITC` for this purpose.

When executing a module containing C code the C runtime libraries have to be included in the `GLOBAL` library lists. The `CERNLIB` command is the most convenient call to this. Note that the C compiler itself is a C program and will abort with an unintelligible message if the libraries are not defined.

VMS

A Fortran program should call `VAXC$CRTL_INIT` (either from Fortran or from C) before calling any functions from the C runtime library. There can be strange effects if this rule is not obeyed.

B.3.2 C main program

Our experience with C main program calling Fortran routines is limited. One known restriction is that the `GETARG` routine for retrieving the command line arguments in Fortran cannot be used. Calling this routine usually results in unresolved externals called `f77_argc` or similar.

Bibliography

[1] R. Brun, C. Kerster, and D. Moffat. *ZCEDEX - Command Interpretation – Creation EDition EXecution* (DD/EE/80-6). CERN, 1980.

[2] R. Hagedorn, J. Reinfelds, C. E. Vandoni, and L. Van Hove. *SIGMA, A New Language for Interactive Array-oriented Computing,* (CERN Report 73-5, 1973 and CERN Report 78-12, 1978. CERN, 1973,1978.

[3] L. Lamport. LaTeX *A Document Preparation System*. Addison-Wesley, 1986.

[4] CN/ASD Group. *PAW users guide,* Program Library Q121. CERN, October 1993.

[5] R. Böck, R. Brun, O. Couet, J. C. Marin, R. Nierhaus, L. Pape, N. Cremel-Somon, C. Vandoni, and P. Zanarini. PAW - Towards a Physics Analysis Workstation. *Computer Physics Communications*, 45:317, 1987.

[6] Open Software Foundation. *OSF/Motif – User's Guide*. OSF, 1981.

[7] CN/ASD Group. *GEANT Detector Description and Simulation Tool, Version 3.16.* CERN, January 1994.

[8] M. Brun, R. Brun, and F. Rademakers. *CMZ - A Source Code Management System*. CodeME S.A.R.L., 1991.

[9] R. Bock et al. *HIGZ Users Guide,* Program Library Q120. CERN, 1991.

[10] D. R. Myers. *GKS/GKS-3D Primer,* DD/US/110. CERN, 1987.

Index