SSN/68.3

Starlink Project
Starlink System Note 68.3

A J Chipperfield

17 December 1997

IFD
Interface Definition Files
1.2

Abstract

Interface Definition Files (IFDs) provide a generic method of defining the interface between Starlink (ADAM) applications and various software environments. Software is described which enables developers to create IFDs and use them to create files required by the Starlink or IRAF environments.

Contents

List of Figures

1 Introduction

The Starlink Software Environment (ADAM) requires that application packages to be run in it have an associated Interface File (`.ifl`) which defines the parameters of the various applications, and package definition files which define the commands available and the source of help *etc.*

Other environments such as IRAF require very similar information presented in a different way.

A file format known as the Interface Definition Format (IFD) has been developed so that a single file can be the source of all the different files required to form the interface between Starlink packages and the software environment in which they are to be run.

This document describes the Interface Definition Format and the software to process IFD files into the required environment-specific files. Currently only ADAM and IRAF are handled. A utility to assist in the production of an IFD file given an ADAM Interface File is also described.

2 Interface Definition Files

2.1 The Basic IFD

Suppose a Starlink application package, PKG, contains two monoliths, pkg_exe1, containing actions act1 and act2, and pkg_exe2, containing actions act3 and act4. The IFD would have the basic form:

```
package pkg {
#  Comments
   executable pkg_exe1 {
      action act1 {
         parameter act1par1 {
           ...
         }
         parameter act1par2 {
           ...
         }
         ...
      }
      action act2 {
         parameter act2par1 {
           ...
         }
         ...
      }
      ...
   }
   executable pkg_exe2 {
      action act3 {
         parameter act3par1 {
           ...
         }
```

```
      ...
   }
   action act4 {
      parameter act4par1 {
        ...
      }
      ...
   }
 }
}
```

where '. . .' represents omitted lines.

This is interpreted by Tcl with the keywords '`package`', '`executable`', '`action`', *etc.* treated as procedures which are defined appropriately depending upon the software environment for which files are being produced. There are currently three scripts which make these definitions and produce environment-specific files:

`ifd2star` Produces files required by the Starlink environment.

`ifd2iraf` Produces files required by the IRAF environment.

`ifd2irafifl` Produces Starlink interface files for use when running Starlink applications from IRAF.

The following points should be noted:

- Keywords must be in lower case.
- Comments may be included – if `#` is found where a command (keyword) is expected, the remainder of the line is treated as comment.
- Lists of values, such as the `vpath` specifier are space-separated (not comma-separated as in ADAM Interface Files).
- String values containing space or `$` must be quoted (with `{}`). Other Tcl special characters will also need to be quoted or escaped.

A 'full' IFD will normally contain additional keywords to define absolutely everything required in producing the environment-specific files.

2.2 Conditional Sections

The IFD may contain sections to be included or excluded depending upon the environment for which it is being processed.

```
environment:  { code }
environment!  { code }
```

The separator '`:`' causes the code to be processed, and '`!`' causes the code to be ignored only if the environment is *`environment`*. In both cases, *`environment`* can be a comma-separated list of environment names and the code may consist of multiple lines.

The term 'environment' is used loosely here – the environment is set to:

`star` when `ifd2star` is running.

`iraf` when `ifd2iraf` is running.

`irafifl` when `ifd2irafifl` is running.

For example with,

```
star:  { code }
```

code will only be processed by `ifd2star`, and with

```
star,iraf!  { code }
```

code will not be processed by either `ifd2star` or `ifd2iraf`.

2.3 Creating IFDs from Interface files

For existing packages a basic IFD can be produced by running the `ifl2ifd` script on the `.ifl` file(s) of the package.

```
% ifl2ifd kappa_mon
```

Will produce IFD `kappa_mon.ifd` from interface file `kappa_mon.ifl`.

Where the package consists of several monoliths, the resulting IFDs must be combined to produce a single IFD.

This basic IFD will define all the package applications which are in the monoliths. However:

(1) Some action definitions may not be required for all environments – the conditional inclusion syntax (see Section 2.2) should be used for these.

(2) Any aliases for the command names should be inserted. This includes the abbreviations allowed in ICL – IRAF has its own system for command abbreviation which does not require any additions to the IFD.

(3) Some parameter definitions may need tweaking. In particular:

- Vector or array parameters which have not been recognised as such (because there was no vector or array static default specified in the interface file) must have a `size` definition added.
- Static defaults should only be specified if they are genuine default values. Some packages do not use the DEFAULT value in the ADAM environment but it is usually used in IRAF and so bad values can cause problems which were not apparent in ADAM. In other cases, no default value has been specified (probably because a GLOBAL value was expected to be used) whereas a default value would be useful for IRAF.

(4) Commands which form part of the package but are not just simple invocations of the applications must be defined.

(5) Any Comments or displays required in the package definition files must be defined.

More information on the changes required for IRAF is given in SSN/35.

3 Producing Starlink Environment Files from an IFD

The **Tcl** script `ifd2star` is used to produce the Starlink environment files for the package.

```
ifd2star package
```

will process IFD *package*`.ifd` and produce the following files required by the Starlink (ADAM) environment:

Interface Files Describing the parameters for the ADAM programs. These files are described in detail in SUN/115. Where applications are combined into monolithic executable files, individual interface files are produced for each application in addition to a monolithic interface file.

The ICL Package Definition File Defining the ICL commands associated with the package and known as the package `.icl` file. These files are described in detail in SSN/64.

Shell Package Definition File Defining the commands associated with running the package direct from the shell. This file is written in two syntaxes, the package `.csh` file for csh-type shells, and a `.sh` file for sh-type shells. These files are described in detail in SSN/64.

4 Producing IRAF Files from an IFD

The **Tcl** scripts `ifd2iraf` and `ifd_irafhlpgen` are used to produce the files required by IRAF. More details on the files produced may be found in SSN/35 and the IRAF documentation.

```
ifd2iraf package
```

will process IFD *package*`.ifd` and produce the following files:

- An IRAF parameter file, *application*`.par`, for each application in the package. These define the IRAF parameters associated with the application. There will be a one-to-one correspondence between the IRAF parameters and the ADAM parameters although they may be handled differently in the two environments.
- An IRAF package definition file, *package*`.cl`. This defines the IRAF commands to run the applications in the package. The name *package* is derived from the argument of the `package` procedure call in the `.ifd` file.
- An IRAF package parameter file, *package*`.par`. This defines parameters for the package - in particular, the version number.
- An Output Parameter File, *executable*`.tcl` for each executable in the package. Each one lists the 'output' and 'dynamic' parameters for each action in the executable. This information is required by the IRAF/Starlink inter-operability system rather than IRAF itself.

and:

```
ifd_irafhlpgen package
```

will read the Starlink HLP source file *package*`.hlp` and produce a set of IRAF help files for the package but it is likely that changes to the text will be required.

5 Producing IRAF-specific Interface Files from an IFD

It may sometimes be found that the interface file required when running a Starlink application from IRAF is different from that required normally. Conditional statements (see Section 2.2) for the '`irafifl`' environment may be included in the IFD and the **Tcl** script `ifd2irafifl` used to create the special interface files (no other files are produced).

For example, with the IFD:

```
package pkg {
  irafifl! { executable pkg_exe1 {
   ...
   }
   executable pkg_exe2 {
      action act1 {
         parameter act1par1 {
            type _REAL
            vpath prompt
          irafifl! {ppath CURRENT DYNAMIC DEFAULT}
          irafifl: {ppath DYNAMIC DEFAULT}
            ...
         }
         ...
      }
      ...
   }
}
```

The command:

```
% ifd2irafifl package
```

will produce an interface file for `pkg_exe2` only, with a `ppath` of '`DYNAMIC,DEFAULT`' for parameter `act1par` of action `act1`.

6 The 'Full' IFD File

The basic keywords in an IFD have already been described but in order to produce complete versions of the required environment files automatically, additional syntax and keywords are available. This section outlines what is available; see elsewhere (Section 7) for complete descriptions of all the keywords.

6.1 IFD Initial Keywords

The following are examples of the additional keywords which may appear within the package description (*i.e.* after `package` *name* {). They are all optional and will be ignored for environments for which they are not relevant.

```
version 1.0              # Specifies a package version number

exepath {$PKGNAME_DIR}   # specifies the directory containing executable
                         # images etc. - default $PKGNAME_DIR

helplib {$PKGNAME_HELP}  # specifies the help library - default
                         # $PKGNAME_HELP
                         # Will be effective until another helplib

prefix pkg               # specifies a prefix for automatic aliases E.g
                         # 'prefix kap' would result in kap_add etc
                         # being defined in addition to add.

display {
Message to display whilst .csh, .sh, .icl scripts for example are running
It will usually be the welcome message for the package.
A display may contain any number of lines which will be displayed line
for line. There may be more that one display in an IFD.
}

defhelp topic entry [library]  # Help on topic may be found in library
                               # (default Current(helplib)) section
                               # defined by entry
```

6.2 Additional Action Keywords

The following are examples of the additional keywords which may appear within the action description (*i.e.* after `action` *name* {). They are all optional and will be ignored for environments for which they are not relevant.

```
alias {acto(ne) act1 act_1} # possible aliases

helplib {$EXAMPLE1_HELP}    # New helplib if required.
```

6.3 Parameter Definition Keywords

6.3.1 General

Within the parameter description (*i.e.* after `parameter` *name* {), The following keywords may appear (example arguments are given):

```
access READ
association <->GLOBAL.COORD_SYSTEM
default {two words}
dynamic yes
```

```
help {\%$KAPPA_HELP ADD PARAMETERS IN1}
helpkey *
in Data World
outputpar
position 1
ppath GLOBAL DYNAMIC
prompt {Co-ordinate system used in the ARD file}
range 1 10
repeated
size *
type _REAL
vpath GLOBAL DEFAULT
```

They are all optional and will be ignored for environments for which they are not relevant. There is an obvious correspondence between most of these keywords and the parameter definition fields of an ADAM interface file as described in SUN/115 which should be consulted for the finer details of permitted values but remember that lists, such the ppath value, are space-separated in the IFD but comma-separated in the Interface File, and character constants only need to be quoted in the IFD (with `{}`) if they contain spaces or `$`.

The keywords which do not have a corresponding ADAM Interface File field are `dynamic`, `outputpar`, `repeated` and `size` (see the following sections).

6.3.2 The `dynamic` keyword

This keyword forces the parameter to be classed as 'dynamic' or 'non-dynamic' regardless of the normal default.

A dynamic parameter is one whose value cannot easily be set as a static default or calculated by the user at runtime. The Starlink Software Environment allows such values to be set by means of VPATH GLOBAL or VPATH DYNAMIC but this is not available for other environments so they must be handled as special cases. In the case of IRAF, for instance, dynamic parameters are handled by forcing the ADAM task to issue a parameter request message with a suggested value. This message is intercepted by the IRAF/Starlink adaptor process which returns the suggested value without asking IRAF for a value. For more information on this, see SSN/35.

In the absence of a `dynamic` keyword, all parameters with VPATH starting with GLOBAL are classed as dynamic and all others (including VPATH DYNAMIC) as non-dynamic.

So that users are warned, particularly when using IRAF `epar`, that changing the parameter is likely to have an unexpected effect, the prompt string has `*!` prepended to it.

6.3.3 The `outputpar` keyword

For non-primitive parameter types, there is a potential confusion between the access required to the parameter and the access required to the file or device whose name is given by the parameter. ADAM requires to be told the access to the file or device and IRAF (more accurately the IRAF/Starlink inter-operability system) needs to know if the value of the parameter itself is output and thus that the IRAF parameter must be updated after the application has run. The list of parameters which must be updated is read from the Output Parameter File created by `ifd2iraf`.

The system will assume that primitive types need updating if the access mode (as defined by the `access` keyword) is 'WRITE' or 'UPDATE' and that other types are not output. For this reason, the `outputpar` keyword is provided to force a non-primitive parameter to be output if necessary.

6.3.4 The `repeated` keyword

This keyword is used to indicate that new values for the parameter may required repeatedly during one invocation of the program. This will usually mean that the user must be prompted each time. In IRAF the recommended default 'automatic' mode allows prompting to be overridden and the same value supplied each time a value is requested so 'query' mode must be set for 'repeated' parameters.

6.3.5 The `size` keyword

ADAM does not require to know the size of a parameter, or even whether it is a scalar, vector or array. However, this information is required for some parameter systems so the `size` keyword is provided in the IFD format.

The IRAF/Starlink inter-operability system only needs to know that the parameter is non-scalar so currently the argument is ignored and may be given as `*`.

6.4 The `command` Keywords

A keyword `command` is provided to define commands for the command language in use (ICL, CL *etc.*) which will do various generic things as follows:

- Define a command which will display a message to the user.

```
command task1 {
  print {task1 has been renamed to task2}
}
```

- Define a command which will obey a command in the underlying shell.

```
command cleanup {
  obey {rm *.tmp}
}
```

- Define a command to run an action with set parameters

```
command fitsexist {
  task fitsmod { taskinherit ndf
                 taskinherit keyword
                 taskparam edit=exist
                 taskparam mode=interface
               }
}
```

  The `taskinherit` subcommand gives the names of parameters of the command whose values will be inherited by the action named in the `task` subcommand. The `taskparam` subcommand gives the name and fixed value of the other parameters to be passed to the action. The code above will result in command `fitsexist` being defined with two parameters, `ndf` and `keyword` so that obeying

```
fitsexist comwest simple
```

is equivalent to obeying `fitsmod` with: parameter `ndf` set to `comwest`; parameter `keyword` set to `simple`; parameter `edit` set to `exist` and parameter `mode` set to `interface`.

Note that in Starlink mode, the csh, sh and ICL user-interfaces will just append anything following the `fitsexist` command to the `fitsmod` command, following the fixed parameters. The `taskinherit` keyword has no effect.

- Define an obsolete command – if it is obeyed, the message is displayed.

```
command oldcommand {
  obsolete {Command oldcommand is obsolete - use newcommand}
}
```

Note that in addition to the subcommand `print`, `obey` *etc.* the command definition may contain `alias` specifications.

6.5 File-specific Output

Sometimes output is required only for one particular file. The following keywords allow lines to be put, verbatim, into the appropriate file. They are ignored if that file is not being produced.

For example:

```
# Output to the package.icl file.
  icl {
\{ This is an ICL comment - NOTE that the brace must be escaped
load file
  }

# Output to the package.csh file.
  csh {
# Define an alias in the .csh file
   alias command shell_command
  }

# Output to the package.sh file.
  sh {
# Define a shell function in the .sh file.
# (the exotic reference to the positional arguments is a portable
# version of plain "$@")
   command () { shell_command ${1+"$@"}; }
  }

# Output to the package.cl file.
  cl {
# Set an IRAF environment variable for the package
set FIGARO_AXES=true
  }
```

7 Details of IFD File Keywords

access
define the access needed to the parameter.

Description:

The access may be READ, WRITE or UPDATE (see SUN/115 for details).

Invocation:

`access` *`mode`*

Arguments:

mode

The required access mode.

Examples:

```
access READ
```

Specifies READ access for the parameter.

Effects:

ADAM: The appropriate `access` field is written to the Interface Files.

IRAF: For file type parameters there is currently no action. For other types the parameter is listed in the Output Parameter File if the access mode is WRITE or UPDATE.

action
Declare an action within an executable

Description:

Declares the name of an action (application) within an executable image and defines the action.

Invocation:

`action` *`actionname`* `{` *`definition`* `}`

Arguments:

actionname

The name of the action.

definition

A Tcl script defining the action in terms of the Tcl procedures declared for an action.

Subcommands:

The following keywords are defined within an `action` definition: `alias`, `parameter`.

Examples:

```
action add {
   parameter in {
      ...
   }
}
```

Defines the action `add` with parameter `in`.

Effects:

ADAM: Opens the individual `.ifl`.

IRAF: Opens the `.par` file.

alias
Define command aliases for the action

Description:

By default a command with the same name as the action will be defined to invoke each action and, if a `prefix` is defined, a command name with the given prefix will also be defined.

The `alias` command allows a list of additional command names to be defined to invoke the action. The list of aliases may be a list of simple command names or names of the form:

com(mand)

where the part before the parentheses is the minimum abbreviation. (Currently this only applies for ICL.)

If a prefix is defined, prefixed command names will also be defined for all aliases.

Invocation:

`alias { *alias_list* }`

Arguments:

alias_list
A list of additional command names.

Examples:

```
alias { acto(ne) act1 }
```

Defines `acto`, `acton` and `act1` as aliases for `actone`.

Effects:

ADAM: As defined.

IRAF: None.

association
Specify an associated GLOBAL parameter

Description:

This corresponds with the ADAM ASSOCIATION field (see SUN/115 for details).

Invocation:

`association` *`specification`*

Arguments:

specification

specifies the name of the associated GLOBAL parameter and the allowed access to it.

Examples:

```
association <->GLOBAL.DEVICE
```

GLOBAL.DEVICE may be used as a source for the value or suggested value of the current parameter, and will be updated with the current value of the parameter if the application ends successfully.

```
association <-GLOBAL.DEVICE
```

GLOBAL.DEVICE may be used as a source for the value or suggested value of the current parameter but will not be updated with the current value of the parameter.

Effects:

ADAM: The appropriate `association` field is written to the Interface Files.

IRAF: None.

cl
Lines for output to the package `.cl` file

Description:

Specifies lines of text to be output to the package `.cl` file.

Invocation:

`cl { text }`

Arguments:

text

The specified text is written to the package `.cl` file. It may consist of more than one line. The text should be legal IRAF command language.

Examples:

```
cl { # Set an IRAF environment variable
   set FIGARO_AXES=true }
```

Inserts a comment and a set command into the package `.cl` file.

Effects:

ADAM: None.

IRAF: As defined.

command
Define a command

Description:

This command defines package commands other than those to run applications in the standard way. The command definition may contain an `alias` subcommand and must contain one of the other subcommands listed below.

Invocation:

`command` *`name`* `{` *`definition`* `}`

Arguments:

name

The name of the command.

definition

A Tcl script defining the command in terms of the Tcl procedures declared for a command.

Subcommands:

The following keywords are defined within a `command` definition:
`alias` – define aliases for the new command
`print` – print a message
`obey` – obey a shell command
`task` – run an action with given parameters
`obsolete` – define an obsolete command.

Examples:

```
command abc {
   alias def
   task xyz { taskparam par=3.0 }
}
```

Defines command `abc` to run command `xyz` with parameter `par` set to 3.0. `def` is an alias for `abc`.

Effects:

ADAM: As specified.

IRAF: As specified.

csh
Lines for output to the `.csh` file

Description:

Specifies lines of text to be output to the `.csh` file.

Invocation:

`csh { *text* }`

Arguments:

text

The specified text is written to the `.csh` file. It may consist of more than one line. The text should be legal C-shell command language.

Examples:

```
csh { # Define the help library
   setenv KAPPA_HELP INSTALL_HELP/kappa }
```

Inserts a comment and a setenv command into the `.csh` file.

Effects:

ADAM: As defined.

IRAF: None.

default
Define the default value for a parameter

Description:

Defines the default value for a parameter and must be of an appropriate type (see SUN/115 for details).

Invocation:

`default` *`value`*

Arguments:

value

The default value for the parameter. It can be an array, in which case the elements should be space-separated.

Examples:

```
default y
```

Specifies TRUE as the default for a LOGICAL parameter.

```
default 1.0 10.0
```

Specifies vector [1.0,10.0] as the default.

```
default {a b}
```

Specifies the string `"a b"` as the default.

```
default a b
```

Specifies the array `["a", "b"]` as the default.

Effects:

ADAM: As specified – output to the `.ifl` files.

IRAF: The value is used as the initial default value in the `.par` file. For primitive data types, a default of `!` is changed to `INDEF`.

defhelp
Define a help topic

Description:

The defhelp procedure specifies the location of help information on a topic assuming a hierarchical help system such as Starlink HLP and the currently defined help library (see `helplib`). Currently `defhelp` is only used in defining help when running from ICL. The location of help on the applications is defined automatically – it is only necessary to include `defhelp` keywords in the IFD for other topics.

Invocation:

`defhelp` *`topic location`*

Arguments:

topic

The topic name.

location

The location within the currently defined help library at which help on the specified topic will be found.

Examples:

```
defhelp data_structures data_structures
```

Help on 'data_structures' is found in subtopic 'data_structures' of the current help library.

```
defhelp kappa 0
```

Help on 'kappa' is found in the top level of the current help library.

Effects:

ADAM: As specified.

IRAF: None.

display
Define a display message

Description:

Defines a message to display whilst `.csh` or `.icl` scripts, for example, are running. It will usually be the welcome message for the package. A display may contain any number of lines which will be displayed line for line. There may be more that one display in an IFD.

Invocation:

`display { *message* }`

Arguments:

message

The message to be displayed. It may consist of more than one line.

Examples:

```
display {
Welcome to the package
          Version 1.1-1
}
```

The message will be displayed, as aligned, when the package is initialised from ICL or the Unix shell.

Effects:

ADAM: As specified.

IRAF: None.

dynamic
Define a parameter to be 'dynamic'

Description:

Forces the parameter to be classed as dynamic or non-dynamic regardless of other considerations. For more information, see 'The `dynamic` Keyword' (Section 6.3.2).

Invocation:

`dynamic` *`switch`*

Arguments:

switch

yes, y, true or t to make the parameter dynamic.
no, n, false or f to make the parameter non-dynamic.

Examples:

```
dynamic yes
```

Forces the parameter to be classed as dynamic.

Effects:

ADAM: None.

IRAF: If the parameter is made dynamic, it is listed in the DynParList array in the Output Parameter File and has its mode set to automatic in the task parameter file and its prompt string will be preceded by `*!`.

Note that parameters with `vpath` starting with `GLOBAL` will default to being dynamic.

executable
Declare an executable image

Description:

Declares the name of an executable image (usually an ADAM monolith) and defines the actions within the image.

Invocation:

`executable` *`image`* `{` *`definition`* `}`

Arguments:

image

The name of the executable image.

definition

A Tcl script defining the executable in terms of the Tcl procedures declared for an executable.

Subcommands:

The following keyword is defined within an `executable` definition: `action`.

Examples:

```
executable kappa_mon { ...  }
```

Define the executable image kappa_mon.

Effects:

ADAM: Opens the monolithic interface file and changes the executable image referred to in the `.icl` and `.csh` files.

IRAF: Opens the `.tcl` file and changes the executable image referred to in the `.cl` file.

exepath
Specify the directory containing executables *etc.*

Description:

The specified directory is used in constructing the Starlink package definition files *package*`.icl` and *package*`.csh`. IT IS NOT USED IN PRODUCING THE IRAF FILES.

If not specified, the directory $*PACKAGE*_DIR is used.

Invocation:

`exepath` *directory*

Arguments:

directory

The directory specification. This could contain environment variable for translation at runtime.

Examples:

```
exepath { $KAPPA_DIR }
```

will result in the directory defined by environment variable KAPPA_DIR being used – for package KAPPA, this is the same as the default.

```
exepath /home/adam4/ajc/kappa
```

will result in directory /home/adam4/ajc/kappa being used.

Effects:

ADAM: As defined.

IRAF: None.

help
Define the 'one-line' help for this parameter

Description:

Currently only for ADAM (see SUN/115 for details).

Invocation:

`helpkey` *`help_specifier`*

Arguments:

help_specifier

The text to be displayed if help is requested or a pointer to a help-file module.

Examples:

```
help { The number of values (between 1 and 10) }
```

Specifies the text to be displayed if parameter help is requested.

```
help { %$KAPPA_DIR ADD PARAMETERS IN1 }
```

Specifies the module in which parameter help is to be found.

Effects:

ADAM: Produces an Interface File `help` field.

IRAF: None.

helpkey
Define the source of help for this parameter

Description:

Currently only for ADAM (see SUN/115 for details).

Invocation:

`helpkey` *`help_specifier`*

Arguments:

help_specifier

A help file and module path.

Examples:

```
helpkey { $KAPPA_HELP PARAMETERS ADD IN1 }
```

Specifies the hierarchy within the help file $KAPPA_HELP at which help on the parameter is found.

```
helpkey *
```

Specifies the default module for help on the parameter.

Effects:

ADAM: Produces an Interface File `helpkey` field.

IRAF: None.

helplib
Specify the pathname of the help library.

Description:

The specified filename is used in constructing ADAM Interface Files and the package definition files package.icl, package.csh and package.sh. IT IS NOT USED IN PRODUCING THE IRAF FILES.

If not specified, the directory $*PACKAGE*_HELP is used.

Invocation:

`helplib` *`library`*

Arguments:

library

The name of the help library. This could contain environment variable for translation at runtime.

Examples:

```
helplib $KAPPA_HELP
```

will result in the file defined by environment variable KAPPA_HELP being used – for package KAPPA, this is the same as the default.

Effects:

ADAM: As specified.

IRAF: None.

icl
Lines for output to the `.icl` file

Description:

Specifies lines of text to be output to the `.icl` file.

Invocation:

`icl { text }`

Arguments:

text

The specified text is written to the `.icl` file. It may consist of more than one line. The text should be legal ICL command language.

Examples:

```
icl { \{ Define a command to print "Hello"
   defstring welcome print "hello" }
```

Inserts a comment and a defstring command into the `.icl` file. NOTE the `{` in the comment must be escaped.

Effects:

ADAM: As defined.

IRAF: None.

in
Define a set of acceptable values for the parameter

Description:

Currently only for ADAM (see SUN/115 for details).

Invocation:

`in` *`set`*

Arguments:

set A list of values of appropriate type.

Examples:

```
in  Red White Blue
```

The acceptable values are: `Red`, `White` or `Blue`.

Effects:

ADAM: Produces an Interface File `in` field.

IRAF: None.

keyword
Specify the name by which the parameter is known to the user.

Description:

Currently only for ADAM and deprecated. This keyword can be used to specify the name by which the parameter is known to the user (on the command line and in prompts and messages *etc*). It defaults to the parameter name.

Invocation:

`keyword { name }`

Arguments:

name

The name to be used.

Examples:

```
parameter x
   keyword y
   ...
}
```

Parameter x will be known as y to the user.

Effects:

ADAM: Produces an Interface File `keyword` field

IRAF: None.

obey
Obey a command language command

Description:

A subcommand of `command`. When the defined command is invoked, a command will be obeyed in the underlying shell. Obviously the command will vary depending upon the shell in use – currently Interface Definition Files assume `csh`.

Invocation:

`obey { *command* }`

Arguments:

command

The command to be obeyed.

Examples:

```
command abc {
   obey { date }
}
```

The command `date` will be obeyed by the underlying shell if command `abc` is obeyed.

Effects:

ADAM: The appropriate commands are written to the `.icl` and `.csh` files.. The implementation for ICL means that the command is actually obeyed by ICL which will pass most shell commands on to the shell.

IRAF: The foreign command mechanism is used.

obsolete
Define an obsolete command

Description:

A subcommand of `command`. If the defined command is invoked, this will usually just print the message – some systems may ignore it altogether.

Invocation:

`obsolete { *message* }`

Arguments:

message

The message to be displayed if the command is obeyed.

Examples:

```
command abc {
   obsolete { Command abc is obsolete - use xyz instead }
}
```

The given message will be displayed if command `abc` is obeyed.

Effects:

ADAM: The appropriate commands are written to the `.icl` and `.csh` files.

IRAF: The foreign command mechanism is used.

outputpar
Force the parameter value to be output

Description:

Forces the parameter to be treated as an 'output' parameter, regardless of the specified access mode. For more information, see 'The outputpar keyword' (Section 6.3.3).

Invocation:

`outputpar`

Arguments:

None

Examples:

```
parameter INFILE {
   type FILE
   access READ
   outputpar
   ...
}
```

The name of the file to be read from is generated by the program and output as the value of parameter INFILE.

Effects:

ADAM: None.

IRAF: The parameter is listed in the Output Parameter File.

package
Define a package

Description:

This command declares the name of a package and defines the commands *etc.* contained in the package.

Invocation:

`package` *`pkgname`* `{` *`definition`* `}`

Arguments:

pkgname

The name of the package. This will be used as the name of the created package definition files.

definition

A Tcl script defining the package in terms of the Tcl procedures declared for a package.

Subcommands:

The following keywords are defined within a `package` definition: `executable`, `version`, `exepath`, `helplib`, `prefix`, `display`, `defhelp`, `command`, `icl`, `csh`, `sh`.

Examples:

```
package kappa {
   executable kappa_mon {
      ...
   }
}
```

Defines the KAPPA package.

Effects:

ADAM: Opens the `.icl` and `.csh` files.

IRAF: Opens the `.cl` file.

parameter
Define an action parameter

Description:

Declares a parameter name and defines its type *etc.*

Invocation:

`parameter` *`name`* `{` *`definition`* `}`

Arguments:

name

The name of the parameter.

definition

A Tcl script defining the parameter in terms of the Tcl procedures declared for a parameter.

Subcommands:

The following keywords are defined within a `parameter` definition: `position`, `type`, `size`, `access`, `outputpar`, `vpath`, `ppath`, `association`, `prompt`, `default`, `in`, `range`, `help`, `helpkey`, `repeated`, `dynamic`. `keyword`.

Examples:

```
parameter par1 {
   position 1
   type _REAL
   ...
}
```

Defines parameter `par1`.

Effects:

ADAM: As specified – output to the `.ifl` files.

IRAF: As specified – output to the `.par` file.

position
Define the command line position for this parameter

Description:

defines a command-line 'position' for the parameter.

Invocation:

`position` *`number`*

Arguments:

number

A command line position.

Examples:

```
action act1
   parameter x
      position 2
      ...
   }
   parameter a
      position 1
      ...
   }
...
}
```

The command `act1 5 10` would invoke action `act1` with parameter `x` set to `10` and parameter `x` set to `5`.

Effects:

ADAM: Produces an Interface File `position` field.

IRAF: Positional parameters are listed first, in the correct order, in the `.par` file.

ppath
Define a search path for the suggested value in a prompt

Description:

This corresponds with the ADAM PPATH field (see SUN/115 for details). The specified search path comprises a space-separated list of one or more of:
CURRENT – The last-used value of the parameter
DEFAULT – take the static default
DYNAMIC – take the dynamic default
GLOBAL – take the value of the associated GLOBAL parameter

Invocation:

`ppath` *`search_path`*

Arguments:

search_path
A space-separated list of possible sources.

Examples:

```
ppath GLOBAL DEFAULT
```

If the associated GLOBAL parameter is not set, use the static default.

Effects:

ADAM: The appropriate `ppath` field is written to the Interface File.

IRAF: None.

prefix
Define a command-name prefix.

Description:

This procedure defines a prefix which may be added to command names in the event of ambiguities between command names in different packages. It will normally be the first three letters of the package name.

Invocation:

`prefix` *`prefix`*

Arguments:

prefix

The prefix to be used.

Examples:

```
prefix kap
```

'kap' will be used as the optional command name prefix.

Effects:

ADAM: As specified.

IRAF: None.

print
Define a command to print a message

Description:

A subcommand of `command`. When the defined command is invoked, the specified message is displayed to the user.

Invocation:

`print { message }`

Arguments:

message

The message to be displayed.

Examples:

```
command abc {
   print { The XXX application is not available for IRAF. }
}
```

The specified text will be displayed if the command `abc` is obeyed.

Effects:

ADAM: The appropriate commands are written to the `.icl` and `.csh` files.

IRAF: The foreign command mechanism is used.

prompt
Specify the prompt string

Description:

Specifies the string to be used when prompting for the parameter. Starlink user-interfaces will usually also display the parameter name and offer a suggested value.

Invocation:

`prompt { *text* }`

Arguments:

text

The prompt string.

Examples:

```
prompt { Type a REAL number }
```

If a value for the parameter is required from the user, a prompt with the specified text will be given.

Effects:

ADAM: The appropriate `prompt` field is written to the Interface Files.

IRAF: The appropriate `prompt` field is written to the `.par` file.

range
Define the range of values permitted for the parameter

Description:

Currently only for ADAM (see SUN/115 for details).

Invocation:

`range` *`min max`*

Arguments:

min

The minimum acceptable value.

max

The maximum acceptable value.

Examples:

```
range 1 10
```

The value must lie between 1 and 10 inclusive.

```
range A Z
```

The value must lie between A and Z inclusive.

Effects:

ADAM: Produces an Interface File `range` field.

IRAF: Set min and max values in the `.par` file.

repeated
Parameter value is obtained repeatedly.

Description:

Informs the system that new values for the parameter may be requested repeatedly during one invocation of the program.

Invocation:

```
repeated
```

Arguments:

None

Examples:

```
parameter INFILE {
   type FILE
   access READ
   repeated
   ...
}
```

There may be repeated request for a new value of the parameter INFILE.

Effects:

ADAM: None.

IRAF: The parameter is set to 'query' mode.

sh
Lines for output to the `.sh` file

Description:

Specifies lines of text to be output to the `.sh` file.

Invocation:

`sh { text }`

Arguments:

text

The specified text is written to the `.sh` file. It may consist of more than one line. The text should be legal sh-style shell command language.

Examples:

```
sh { # Define the help library
   KAPPA_HELP=INSTALL_HELP/kappa; export KAPPA_HELP
}
```

Inserts a comment, and a command to set an environment variable, into the `.sh` file.

Effects:

ADAM: As defined.

IRAF: None.

size
Define the size of the parameter

Description:

The parameter size may be given as any string (usually *). The actual value is not used but some systems needs to know if the parameter is a vector or array.

Invocation:

`size` *`size`*

Arguments:

size

Any string to indicate the size.

Examples:

```
size *
```

Specifies that the parameter takes an array value.

```
size 2
```

Specifies that the parameter takes an array value.

Effects:

ADAM: None.

IRAF: The parameter is defined as type struct.

task
Invoke an action with a given set of parameters.

Description:

A subcommand of `command`. When the defined command is invoked, it invokes the named action. Fixed and variable parameters for the action may be specified. The named action must be in the current package and already be defined.

Invocation:

`task` *`name`* `{` *`description`* `}`

Arguments:

name
The name of the action to be invoked.

description
A list of parameter definitions using the `taskparam` and `taskinherit` keywords.

Examples:

```
command abc {
   task xyz {
      taskparam {method=list}
      taskparam {value=1.0}
      taskinherit ndf
}
```

Obeying command `abc filename` will be equivalent to obeying command `xyz` with parameters `method` and `value` set as specified and parameter `ndf` set to `filename`.

Effects:

ADAM: The appropriate command definitions are written to the `.icl` and `.csh` files. Note that the `taskinherit` keyword has no effect. Anything following the primary command invocation will be appended to the invocation of the named action, following the fixed parameters.

IRAF: A CL procedure is created and a command defined in the package `.cl` file to run it. The procedure will have parameters as defined by any `taskinherit` keywords and will invoke the named action with the inherited parameter values followed by the fixed parameters, all in `keyword=value` form.

taskinherit
Define a parameter name for a `command task` command

Description:

A subcommand of `task`. Specifies the name of a parameter of the action whose value is to be inherited from the top-level command

Invocation:

`taskinherit { parameter_name }`

Arguments:

parameter_name
The name of the parameter

Examples:

See `task` example.

Effects:

ADAM: None – anything following the command name on the command line will be added, verbatim, to the action invocation.

IRAF: The created task procedure (see `task`) has a parameter with the given name which is inherited by the secondary command.

taskparam
Define a parameter value for a `command task` command

Description:

A subcommand of `task`. The parameter specification is added, verbatim, to the invocation of the name action.

Invocation:

`taskparam { *parameter_specification* }`

Arguments:

parameter_specification

A string to be added to the command line when invoking the named action. It may be any string which is legal as a parameter of the command specified by the `task` keyword but will normally be of the form '`keyword=value`'.

Examples:

See `task` example.

Effects:

ADAM: The parameter specification is added to the command written to the `.icl` and `.csh` files.

IRAF: The parameter specification is added to the command line in the created task procedure (see `task`) which invokes the action.

type
Define the type of the parameter

Description:

The parameter type may be: _CHAR, LITERAL, _DOUBLE, _INTEGER, _REAL, _LOGICAL or a structure type (see SUN/115 for details). For environments other than ADAM, suitable choices are made.

Invocation:

`type` *`type`*

Arguments:

type

The ADAM type of the parameter.

Examples:

```
type _REAL
```

Specified type _REAL for the parameter.

Effects:

ADAM: As specified – output to the `.ifl` files.

IRAF: Scalar _CHAR, LITERAL, _DOUBLE, _INTEGER, _REAL and _LOGICAL types become string, string, real, integer, real and boolean respectively. Vectors and arrays are defined as type struct (see the `size` keyword). All other types are treated as filenames.

version
Define the version number of the package.

Description:

This is optional. The preferred way of setting the version number into package files is to use the PKG_VERS macro in the makefile at install time.

Invocation:

`version` *`version_number`*

Arguments:

version_number

The package version number.

Examples:

```
version V2.1-1
```

Specifies version 2.1-1

Effects:

ADAM: None.

IRAF: If defined the version number will be written as the value of the version parameter in the *package*`.par` file. If it is not defined, it defaults to '`PKG_VERS`'.

vpath
Define the source of the parameter value.

Description:

This corresponds with the ADAM VPATH field (see SUN/115 for details). The specified search path comprises a space-separated list of one or more of:
CURRENT – The last-used value of the parameter
PROMPT – prompt for the value
DEFAULT – take the static default
DYNAMIC – take the dynamic default
GLOBAL – take the value of the associated GLOBAL parameter
NOPROMPT – prevents a prompt as a last resort.
INTERNAL – can only be used alone, implies DYNAMIC,CURRENT,NOPROMPT.

Invocation:

`vpath` *`search_path`*

Arguments:

search_path
A space-separated list of possible sources.

Examples:

```
vpath CURRENT DEFAULT
```

If there is no current value, use the static default.

Effects:

ADAM: The appropriate `vpath` field is written to the Interface Files.

IRAF: In the absence of a `repeated` or `dynamic` keyword, the following rules apply: If the first element of the vpath is `PROMPT` or there is no vpath specifier, the IRAF parameter is made automatic mode. If the first element is `GLOBAL`, the parameter is made dynamic. In all other cases it is made hidden.

The `repeated` or `dynamic` keywords will override this behaviour.