```
clpackage.language:
Language package.
Note: To get help on a keyword enclose it in quotes.  Keywords are starred.
          intro - A brief introduction to IRAF

Language components:

          break * Break out of a loop
           case * One setting of a switch
       commands - A discussion of the syntax of IRAF commands
        cursors - Graphics and image display cursors
   declarations - Parameter/variable declarations
        default * The default clause of a switch
           else * Else clause of IF statement
            for * C-style for loop construct
             if * If statement
           goto * Goto statement
        logging - Discussion of CL logging
           next * Start next iteration of a loop
     parameters - Discussion of parameter attributes
      procedure * Start a procedure script
         return * Return from script with an optional value
         switch * Multiway branch construct
          while * While loop


Builtin Commands and Functions:

         access - Test if a file exists
           back - Return to the previous directory (after a chdir)
           beep - Send a beep to the terminal
            bye - Exit a task or package
          cache - Cache parameter files, or print the current cache list
             cd - Change directory
          chdir - Change directory
             cl - Execute commands from the standard input
          clbye - A cl followed by a bye (used to save file descriptors)
          clear - Clear the terminal screen
         defpac - Test if a package is defined
         defpar - Test if a parameter is defined
        deftask - Test if a task is defined
         dparam - Dump a pset as a series of task.param=value assignments
           edit - Edit a text file
       ehistory - Edit history file to re-execute commands
         envget - Get the string value of an environment variable
         eparam - Edit parameters of a task
          error - Print error code and message and abort
      flprcache - Flush the process cache
         fprint * Print a line into a parameter
          fscan * Scan a list
         gflush - Flush any buffered graphics output
       hidetask - Make a task invisible to the user
        history - Display  commands previously executed
           jobs - Display status of background jobs
           keep - Make recent set, task, etc. declarations permanent
           kill - Kill a background job
         logout - Log out of the CL
         lparam - List the parameters of a task
       mathfcns - Mathematical routines
         mktemp - Make a temporary (unique) file name
           osfn - Return the host system equivalent of an IRAF filename
        package - Define a new package, or print the current package names
        prcache - Show process cache, or lock a process into the cache
          print - Format and print a line on the standard output
         putlog - Put a message to the logfile
          radix - Encode a number in the specified radix
       redefine - Redefine a task
          reset - Reset the value of an environment variable
           scan * Scan the standard input
        service - Service a query from a background job
            set - Set an environment variable
           show - Show an environment variable
          sleep - Hibernate for a specified time
        strings - String manipulation routines
           stty - Set/show terminal characteristics
           task - Define a new task
           time - Print the current time
        unlearn - Restore the default parameters for a task or package
         update - Update a task’s parameters (flush to disk)
           wait - Wait for all background jobs to complete
```

NAME
access -- test whether a file exists

USAGE
bool = access (filename)

PARAMETERS

filename
    The name of the file whose existence is to be tested.

DESCRIPTION
ACCESS is a boolean intrinsic function returning true ("yes") if the named file exists. ACCESS can only be called as a function in an expression, not as a task.

EXAMPLES
1. Type a file if it exists.

```
if (access ("lib$motd"))
    type ("lib$motd")
else
    error (11, "File not found")
```

2. Tell if a file exists.

```
    cl> = access ("lib$motd")
```

BUGS
An optional second argument should be added to test whether the named file can be accessed for reading or writing.

NAME
back -- return to the previous directory

USAGE
back

PARAMETERS
None.

DESCRIPTION
BACK is used after a call to CHDIR or CD to return to the previous directory. Repetitive calls to BACK may be used to toggle between two directories.

EXAMPLES
1. Go to the logical directory "dataio".

```
    cl> cd dataio
```

2. Return to the previous directory, and then go back to the dataio directory.

```
    cl> back;back
```

SEE ALSO
chdir, pathnames

NAME
    beep -- beep the terminal

USAGE
    beep

DESCRIPTION
    Beep sends the bell character (^G) to the terminal.

EXAMPLES
    1. Wait for a background job to complete, ringing the terminal bell when done.

```
cl> wait;beep
```

SEE ALSO
    clear

NAME
    break -- break out of a loop

USAGE
    break

DESCRIPTION
    The BREAK statement is used to exit (break out of) the FOR or WHILE loop in which it is found. In the case of nested loop constructs only the innermost loop is terminated. Unlike C usage the BREAK statement does not break out of a switch.

EXAMPLES
    1. Scan a list (file), printing each list element until either the list is exhausted or a list element "exit" or "quit" is encountered.

```
while (fscan (list, s1) != EOF) {
    if (s1 == "exit" || s1 == "quit")
        break
    print (s1)
}
```

    2. Sum the pixels in a two dimensional array, terminating the sum for each line if a negative pixel is encountered, and terminating the entire process when the total sum passes a predefined limit.

```
total = 0
for (i=1;  i <= NCOLS;  i+=1) {
    for (j=1;  j <= NLINES;  j+=1) {
        if (pixel[i,j] < 0)
            break               # exit the J loop
        total += pixel[i,j]
    }
    if (total > NPHOT)
        break                   # exit the I loop
}
```

BUGS

SEE ALSO
    next, while, for

NAME
bye -- terminate task execution

USAGE
bye

DESCRIPTION
The BYE command terminates the task from which it is executed. This is exactly equivalent to the CL reading end of file (EOF) when executing a task.

EXAMPLES
1. The most common usage of BYE occurs when it is typed by the user to exit a package; in this case, BYE terminates the package script task.

```
cl> plot
pl> bye
cl>
```

SEE ALSO
clbye, return

NAME
cache -- cache the parameters for a task in fast memory

USAGE
cache task [task ...]

PARAMETERS

task
: The name of a task whose parameter set is to be cached in fast memory.

DESCRIPTION
The CACHE command loads the parameters of a task in memory. The CL normally reads the parameters for a task from disk whenever the task is executed. Cacheing the parameters for frequently executed tasks can speed up execution significantly. This is particularly important when the tasks are called from within a loop.

If the CACHE command is entered without any arguments a list of the currently "cached" tasks is printed.

EXAMPLES
1. Cache the parameters for the tasks DIRECTORY and PAGE.

```
cl> cache dir page
```

2. Cache the parameters for the tasks called in a loop within the body of a procedure script. Note the use of command mode in the script.

```
begin
        cache ("alpha", "beta")
        for (i=1;  i <= 10;  i+=1) {
            alpha (i)
            beta (i)
        }
end
```

BUGS
The parameter cache should not be confused with the process cache associated with the PRCACHE and FLPRCACHE commands.

SEE ALSO
unlearn, update, lparam, eparam

NAME

switch -- switch case statement

SYNTAX

```
switch (expr) {
case val1 [, val1,...]:
    statements
case val3 [, val3,...]:
    statements
        (etc.)
default:
    statements
}
```

ELEMENTS

expr
An integer-valued expression tested before entry into the switch block.

valN
Integer valued constants used to match expression.

statements
Simple or compound statements to be executed when the appropriate case or default block is selected.

DESCRIPTION

The SWITCH statement provides a multiway branch capability. The switch expression is evaluated and control branches to the matching CASE block. If there is no match the DEFAULT block, if present, receives control. If no DEFAULT block is present, the switch is skipped.

Each CASE statement consists of a list of values defining the case, and an executable statement (possibly compound) to be executed if the case is selected by the switch. Execution will continue until the next case is reached, at which time a branch out of the SWITCH statement occurs. Note this difference from the C switch case, where an explicit BREAK statement is required to exit a switch. If a BREAK is used in a CL switch, it will act upon the loop statement containing the switch, not the switch itself.

Note that both the switch expression and the case constants may be integers, or single characters which are evaluated to their ASCII equivalents.

The DEFAULT statement must be the last statement in the switch block.

EXAMPLES

1. Multiple cases, no default case.

```
switch (opcode) {
case 1:
    task1 (args)
case 2:
    task2 (args)
case 5:
    task5 (args)
}
```

2. Multiple values in a case.

```
switch (digit) {
case '1','2','3','4','5','6','7':
    n = n * 8 + digit - '0'
default:
    error (1, "invalid number")
}
```

BUGS

Only integer values are allowed (no strings). The case values must be constants; ranges are not permitted.

SEE ALSO

if else, goto

NAME
    chdir, cd -- change the current working directory

USAGE
    chdir [newdir] or cd [newdir]

PARAMETERS

newdir
    The new working directory. The special name "." refers to the current directory; ".." refers to the next higher directory.

DESCRIPTION
    CHDIR is used to change the current working directory. When called without any arguments, CHDIR sets the default directory to "home$", the users home directory. The new directory can be specified as an IRAF logical name, as a sub-directory of the current directory, as a path from either a logical directory or the current directory, or as an operating system dependent name.

    The names CHDIR and CD are synonyms. Note that the command BACK may be called after a CHDIR to return to the previous directory without typing its name.

EXAMPLES
    1. Return to our home directory.

```
cl> cd
```

    2. Go to the package logical directory "pkg$".

```
cl> chdir pkg
```

    3. Go down one level to the directory "dataio", a subdirectory of "pkg".

```
cl> cd dataio
```

    4. From "dataio", go back up to "pkg" and down into "images".

```
cl> cd ../images
```

    5. Go to the "tv" directory, a subdirectory of "images", regardless of the current directory

```
cl> cd pkg$images/tv
```

    6. On a VMS system, define a new logical directory on a different

disk device and go there. Note that the character $ is not permitted in host file or directory names.

```
cl> set dd = scr1:[data]
cl> cd dd
```

SEE ALSO
    back, pathnames

NAME
    chdir, cd -- change the current working directory

USAGE
    chdir [newdir] or cd [newdir]

PARAMETERS

    newdir
        The new working directory. The special name "." refers to the current directory; ".." refers to the next higher directory.

DESCRIPTION
    CHDIR is used to change the current working directory. When called without any arguments, CHDIR sets the default directory to "home$", the users home directory. The new directory can be specified as an IRAF logical name, as a sub-directory of the current directory, as a path from either a logical directory or the current directory, or as an operating system dependent name.

    The names CHDIR and CD are synonyms. Note that the command BACK may be called after a CHDIR to return to the previous directory without typing its name.

EXAMPLES
    1. Return to our home directory.

```
cl> cd
```

    2. Go to the package logical directory "pkg$".

```
cl> chdir pkg
```

    3. Go down one level to the directory "dataio", a subdirectory of "pkg".

```
cl> cd dataio
```

    4. From "dataio", go back up to "pkg" and down into "images".

```
cl> cd ../images
```

    5. Go to the "tv" directory, a subdirectory of "images", regardless of the current directory

```
cl> cd pkg$images/tv
```

    6. On a VMS system, define a new logical directory on a different

disk device and go there. Note that the character $ is not permitted in host file or directory names.

```
cl> set dd = scr1:[data]
cl> cd dd
```

SEE ALSO
    back, pathnames

NAME
    cl   -- call the CL as a task
    clbye -- like cl(), but closes current script file too

PARAMETERS

gcur = ""
    Global graphics cursor.

imcur = ""
    Global image cursor.

abbreviate = yes
    Permits minimum match abbreviations of task and parameter names (disabled within scripts).

echo = no
    Echo all commands received by the CL on the terminal.

ehinit = "standout eol noverify"
    Ehistory options string. (See "ehistory")

epinit = "standout noshowall"
    Eparam options string. (See "eparam")

keeplog = no
    Keep a log of all CL commands.

logfile = "uparm$logfile"
    The name of the logfile, if command logging is enabled.

logmode = "commands nobackground noerrors notrace"
    Logging mode control parameter. (See "logging")

lexmodes = yes
    Enable automatic mode switching between "command mode" (used when commands are being entered interactively at the terminal), and "compute mode" (used to evaluate arithmetic expressions and argument lists). If LEXMODES is disabled command mode is disabled. Command mode is always disabled within scripts and within parenthesis, i.e., expressions or formal argument lists.

menus = yes
    If MENUS are enabled, a table will be printed whenever a package is entered or exited listing the tasks (or subpackages) in the new package.

mode = "ql"
    The parameter mode of the CL, and of any tasks run by the CL which do not specify their own mode (i.e., which specify ‘auto’ mode). A "q" causes a query to be generated whenever a

parameter is used which was not set explicitly on the command line. An "m" (menu mode) causes EPARAM to be called to edit/check a task’s parameters when the task is run interactively. An "l" causes the parameter file for a task to be updated on disk whenever the task is run interactively. Note that changing the mode at the CL level will have no affect on the operation of an individual task unless "auto" mode is set at the package, task, and parameter level, causing the mode to defer to the global CL mode.

notify = yes
    If NOTIFY is enabled background jobs will print a message on the user terminal (or in the logfile for a queued job) notifying the user when the job completes.

szprcache = (a small number)
    Controls the size of the process cache. The value may range from 1 to 10. A larger number reduces process spawns but the idle processes may consume critical system/job resources.

DESCRIPTION

The CL and CLBYE commands are used to call the CL as a task. The function of the CL task is to read and execute commands from its standard input until BYE or end of file is reached. The CL task may be called with arguments or executed in the background like any other task. The CL task may be called from within a procedure or script to read commands from the command stream which called that procedure or task; this is usually the terminal but may be a another script.

When the CL or CLBYE command is invoked, the command language interpreter stores information about which tasks and packages are currently defined. When the command is finished any tasks or packages which have become defined since invocation are lost, unless the user specifically overrides this by using the KEEP command.

The CLBYE command performs exactly like a CL followed by a BYE, except that when called from a script the script file is closed immediately, freeing its file descriptor for use elsewhere. If CL is used instead of CLBYE in a script, the file is not closed until after the CL returns. If a CLBYE is used in a script, any commands following the CLBYE will not be executed.

EXAMPLES

1. Execute CL commands from a file.

```
cl> cl < cmdfile
```

2. Execute CL commands from a pipe.

```
        cl> print ("!type ", fname) | cl
```

3. Execute CL, taking command input from the terminal. Since command input is already from the terminal, the only effect is to mark the state of CL memory, to allow TASK, SET, and other definitions to be made temporarily and later freed by terminating the CL with a BYE.

```
        cl> cl
        cl> set pak = "home$tasks/"
        cl> task $mytask = pak$x_mytask.e
                (execute the task)
        cl> bye
```

In the example above, the declarations of the logical directory "pak" and the task "mytask" are discarded when the BYE is entered, terminating the CL.

BUGS

Beware that any changes made to the global CL parameters during the execution of a CL remain in effect after the task terminates.

SEE ALSO

bye, keep, logout

NAME

cl -- call the CL as a task
clbye -- like cl(), but closes current script file too

PARAMETERS

gcur = ""
: Global graphics cursor.

imcur = ""
: Global image cursor.

abbreviate = yes
: Permits minimum match abbreviations of task and parameter names (disabled within scripts).

echo = no
: Echo all commands received by the CL on the terminal.

ehinit = "standout eol noverify"
: Ehistory options string. (See "ehistory")

epinit = "standout noshowall"
: Eparam options string. (See "eparam")

keeplog = no
: Keep a log of all CL commands.

logfile = "uparm$logfile"
: The name of the logfile, if command logging is enabled.

logmode = "commands nobackground noerrors notrace"
: Logging mode control parameter. (See "logging")

lexmodes = yes
: Enable automatic mode switching between "command mode" (used when commands are being entered interactively at the terminal), and "compute mode" (used to evaluate arithmetic expressions and argument lists). If LEXMODES is disabled command mode is disabled. Command mode is always disabled within scripts and within parenthesis, i.e., expressions or formal argument lists.

menus = yes
: If MENUS are enabled, a table will be printed whenever a package is entered or exited listing the tasks (or subpackages) in the new package.

mode = "ql"
: The parameter mode of the CL, and of any tasks run by the CL which do not specify their own mode (i.e., which specify ‘auto’ mode). A "q" causes a query to be generated whenever a

parameter is used which was not set explicitly on the command line. An "m" (menu mode) causes EPARAM to be called to edit/check a task's parameters when the task is run interactively. An "l" causes the parameter file for a task to be updated on disk whenever the task is run interactively. Note that changing the mode at the CL level will have no affect on the operation of an individual task unless "auto" mode is set at the package, task, and parameter level, causing the mode to defer to the global CL mode.

notify = yes
: If NOTIFY is enabled background jobs will print a message on the user terminal (or in the logfile for a queued job) notifying the user when the job completes.

szprcache = (a small number)
: Controls the size of the process cache. The value may range from 1 to 10. A larger number reduces process spawns but the idle processes may consume critical system/job resources.

DESCRIPTION

The CL and CLBYE commands are used to call the CL as a task. The function of the CL task is to read and execute commands from its standard input until BYE or end of file is reached. The CL task may be called with arguments or executed in the background like any other task. The CL task may be called from within a procedure or script to read commands from the command stream which called that procedure or task; this is usually the terminal but may be a another script.

When the CL or CLBYE command is invoked, the command language interpreter stores information about which tasks and packages are currently defined. When the command is finished any tasks or packages which have become defined since invocation are lost, unless the user specifically overrides this by using the KEEP command.

The CLBYE command performs exactly like a CL followed by a BYE, except that when called from a script the script file is closed immediately, freeing its file descriptor for use elsewhere. If CL is used instead of CLBYE in a script, the file is not closed until after the CL returns. If a CLBYE is used in a script, any commands following the CLBYE will not be executed.

EXAMPLES

1. Execute CL commands from a file.

```
cl> cl < cmdfile
```

2. Execute CL commands from a pipe.


```
cl> print ("!type ", fname) | cl
```

3. Execute CL, taking command input from the terminal. Since command input is already from the terminal, the only effect is to mark the state of CL memory, to allow TASK, SET, and other definitions to be made temporarily and later freed by terminating the CL with a BYE.

```
cl> cl
cl> set pak = "home$tasks/"
cl> task $mytask = pak$x_mytask.e
        (execute the task)
cl> bye
```

In the example above, the declarations of the logical directory "pak" and the task "mytask" are discarded when the BYE is entered, terminating the CL.

BUGS

Beware that any changes made to the global CL parameters during the execution of a CL remain in effect after the task terminates.

SEE ALSO

bye, keep, logout

NAME
    clear -- clear the terminal screen

USAGE
    clear

DESCRIPTION
    The CLEAR command clears the terminal screen. For this to work properly the environment variable TERMINAL must correctly identify the terminal currently in use. If the terminal should get stuck in reverse video mode, CLEAR will restore normal video mode as well as clearing the screen.

EXAMPLES
    1. Clear the screen and print the current directory.

```
        cl> cle;dir
```

SEE ALSO
    beep, stty

NAME
    commands -- the CL command syntax

SYNTAX
    In COMMAND mode (normal interactive commands):

```
        taskname arg1 ... argN par=val ... par=val redir
```

    In COMPUTE mode (algebraic mode, for expressions and procedures)

```
        taskname (arg1, ... argN, par=val, ... par=val redir)
```

ELEMENTS

    taskname
        The name of the task to be executed.

    argN
        The positional arguments to the task. An argument may be any expression; in command mode, a parameter name must be enclosed in parenthesis to avoid interpretation as a string constant (e.g., filename).

    param=value
        Keyword equals value assignment. The value of the parameter named on the left is set equal to the value of the expression on the right. Keyword equals value assignments must follow any positional arguments. To save typing, boolean (yes/no) parameters may be set with a trailing + or -, e.g., "verbose+" is the same as "verbose=yes".

    redir
        A file redirection argument, e.g.:

```
        > file          spool output in a file
        < file          read input from a file (rather than the terminal)
        >> file         append the output to a file
        >& file         spool both error and regular output in a file
        >>& file        append both error and regular output to a file
        >[GIP]          redirect graphics output to a file, e.g, >G file
        >>[GIP]         append graphics output to a file, e.g, >G file
```

DESCRIPTION
    A CL command is an invocation of a predefined CL task. A task may be one of the numerous builtin functions (e.g. time, lparam, ehistory), a task defined in a package supplied to the user automatically, (e.g., the DIRECTORY task in the SYSTEM package), or a task the user has defined himself, using the TASK directive.

The entire argument list of a command, including file redirection arguments may be enclosed in parentheses. This forces all arguments to be evaluated in compute mode. In command mode arguments are delimited by spaces and most characters may be included in strings without need to quote the strings. In compute mode arguments are delimited by commas, all strings must be quoted, and all CL arithmetic and other operators are recognized. Command mode is the default everywhere, except within parenthesis, on the right hand side of a "= expr" (inspect statement), or within procedures. The sequence #{ <statements> #} may be used to force interpretation of a series of statements in compute mode.

1. ARGUMENTS

The task name may be followed by any number of positional arguments and/or keyword=value type arguments, switches, or i/o redirection arguments. The positional arguments must come first. Arguments are most commonly simple numeric or string constants, but general expressions are allowed. Some examples of arguments follow.

```
"quoted string"
(cos(.5)**2 + sin(.5)**2)
"work" // 02
k + 2                   # valid only in compute mode
i+3                     # valid in both modes
(i+3)                   # same answer in both modes
```

Within an argument the treatment of unquoted strings depends upon the current mode. In command mode the string is assumed to be a string constant, while in compute mode it is taken to be the name of a parameter. For example, in command mode the expression

```
i+3
```

is equivalent to the string "i+3", while in compute mode this would evaluate to the sum of the VALUE of the parameter "i" plus 3. To force evaluation of a string like i+3 as a arithmetic expression, enclose it in parenthesis.

Positional arguments are assigned to the parameters of the task to be executed. The position of each task parameter is determined by the order of the arguments in the PROCEDURE declaration of a procedure script task, or by the order of declaration of the parameters in a parameter file for other tasks.

Hidden parameters cannot be assigned values positionally (one must use keywork assignment). It is an error to have more positional arguments than there are corresponding parameters in the task, but omitting positional arguments is legal. In compute mode, arguments may be skipped using commas to mark the skipped arguments, e.g. a,,b.

Following the positional arguments the user may specify keyword arguments. All parameters of a task, including hidden parameters may be assigned to using keyword arguments. The form of a keyword argument is

```
param=expr
```

where PARAM is the name of the task's parameter, and EXPR is any legal CL expression. If the parameter is boolean an alternative syntax called the "switch" syntax is available:

```
param+          # same as param=yes
param-          # same as param=no
```

A given parameter may only be assigned to once in a command line.

2. I/O REDIRECTION

Following the argument list the user may specify one or more file redirection parameters. This permits the altering of standard i/o streams for this command only. Note that the file name specified is interpreted according to the current mode, i.e.

```
> file
```

sends output to a file with the name "file" in command mode, but uses the VALUE of the parameter "file" as the filename in compute mode.

The output from one command may also be directed to the input of another using pipes. The syntax is

```
    command1 | command2
or
    command1 |& command2
```

Here command1 and command2 are full commands, including the taskname and all arguments. In the first example the standard output of command1 becomes the standard input of command2, while in the second the both the standard and error output are sent to command2.

Once two commands have been joined by a pipe they function effectively as a single command, and the combined command may be joined by pipe to further commands. The resulting "command block" is executed as a unit, and may be submitted as a background job by following the command block with an "&" or "& queuename".

EXAMPLES

1. Simple positional arguments only (command mode).

```
cl> copy file1 file2
```

2. Simple positional arguments only (compute mode).

```
cl> copy ("file1", "file2")
```

3. One positional argument, i.e., the string "file1,file", and one keyword=value type argument. Note that string need not be quoted even though it contains the comma, provided there are no spaces in the string.

```
cl> lprint file1,file2 device=versatec
```

4. Syntax for i/o redirection in compute mode, as in a script.

```
type ("*.x", > "spool")
```

5. The same command in command mode.

```
cl> type *.x > spool
```

6. Use of an arithmetic expression in command mode; the scalar value of the expression given as the third positional argument is added to the value of every pixel in image "pix1", writing a new image "pix2" as output.

```
cl> imarith pix1 + (log(4.2)+10) pix2
```

Many additional examples may be found in the EXAMPLES section of the manual pages throughout the system.

SEE ALSO

procedure, parameters

NAME

cursors -- cursor control for graphics and image display devices

1. INTRODUCTION

In IRAF, all cursor input is via the graphics cursor interface in the CL. The CL supports two types of cursors, the graphics cursor and the image display cursor, represented in the CL by the two parameter datatypes GCUR and IMCUR. To read either cursor from a task, the programmer declares one of the parameters of their task to be of type gcur or imcur, and then reads the value of the parameter. The act of reading a cursor type parameter causes the physical device cursor to be read. To make it possible for the user to read either cursor at any time, the CL provides two predefined global parameters also called GCUR and IMCUR (or to be more precise, "cl.gcur" and "cl.imcur", since both parameters are local parameters of the CL task).

Since the graphics cursors are interfaced as CL parameters, a cursor read is implied whenever a cursor type parameter is referenced in a CL expression. The simplest way to read a cursor is to use the "inspect" statement to print the value of the parameter, as in the following example. Exactly the same thing happens when a program like IMPLOT or SPLOT reads the cursor.

```
cl> = gcur
0.5005035 0.4980621 1 k
```

More complex accesses are occasionally useful, e.g.:

```
cl> print (gcur, > "curpos")
```

writes the cursor value into a file, and

```
cl> = fscan (gcur, x, y)
```

leaves the X and Y coordinates of the cursor in parameters X and Y.

A cursor read returns a string value, as can be seen in the above example. The fields of the cursor value string are (from the left) the X and Y position of the cursor in world coordinates, the number of the world coordinate system to which the coordinates refer, and the key value, or character typed to terminate the cursor read. If the key is a colon (":"), a fourth field will be added, namely a character string entered by the user after typing the colon key. This feature is useful for passing arbitrary commands to programs via the cursor interface.

A cursor read is not instantaneous. A cursor read is initiated by reading a cursor type parameter, and terminated by typing any lower case or nonalphanumeric character on the keyboard. The keyboard is used to terminate cursor reads from the image display as well as from a graphics terminal. While the cursor read is in progress, i.e., while the CL is waiting for a key to be typed on the terminal, the CL is said to be in CURSOR MODE. Cursor mode reserves all of the upper case characters and digits for cursor mode commands. Since the cursor mode commands are intercepted by cursor mode, they do not terminate a cursor read and are never seen by the program reading the cursor.

The cursor mode commands are the major topic of discussion in the remainder of this document. In brief, the cursor mode commands are used to zoom in on some portion of the graphics frame (e.g., to get a more accurate cursor position measurement), to roam about at high magnification, to replot the frame, to make a hardcopy on a batch plotter device, to save or restore the frame in a file, and so on. In reading the rest of this document, take care not to get lost in the complexities of cursor mode, forgetting the essential simplicity of the cursor interface, namely that we are reading the cursor and returning the cursor coordinates to the caller.

In the remainder of this document the discussion will focus on the graphics cursor to minimize confusion. The same interface is however used to access both types of cursor, hence the discussion is relevant for the image display interface as well.

2. OVERVIEW

2.1 INVOKING CURSOR MODE

Many IRAF tasks produce a plot of some sort and then bring up a graphics cursor (e.g. a crosshair) and automatically leave the terminal in cursor mode. Alternatively, the user can invoke cursor mode from the CL by typing:

```
cl> = gcur
```

If the CL environment variable CMINIT is defined when cursor mode is first entered, the string value will be interpreted as a cursor mode command and used for initialization. For example, to speed up drawing time you could set text quality to low and the graphics resolution to 200 points in X and 100 points in Y by adding the following SET declaration to one's "login.cl" file:

```
set cminit = "xres=200; yres=150; txqual=low"
```

An additional environment variable is provided for applications which generate very complex plots. There is a fixed upper limit on the size of the cursor mode frame buffer, used to retain all the graphics instructions used to generate a plot. If the buffer overflows the plot will come out correctly the first time, but part of the instructions used to generate the plot will be discarded, hence it will not be

possible to regenerate the full plot in cursor mode. If this happens the size of the cursor mode frame buffer may be increased, e.g.,

```
set cmbuflen = 512000
```

would set the size of the frame buffer to 512K words, or 1 megabyte. This would be large enough to hold almost any plot. A call to GFLUSH may be required before the new buffer size takes effect.

2.2 CURSOR MODE HELP

While in cursor mode, help text may be obtained in at least two ways. Help on the cursor mode commands themselves, i.e. the topic of this document, is available with the command ":.help" or just ":.". By convention help on an application task running cursor mode, e.g. IMPLOT, is available with the command "?". All interactive IRAF graphics tasks are required to respond to the ? key with a summary of the keystrokes recognized by that task.

2.3 CURSOR MODE COMMANDS AND OPTIONS

While in cursor mode, whether invoked by an IRAF task or interactively via the command "=gcur", three classes of commands are available. First, single, upper-case letters take actions such as roaming and zooming, redrawing axes after a zoom, and prompting for text annotation. Second, cursor mode options and more complicated commands may be entered after a ":.", for example sending a screen snapshot to a hardcopy plotter and changing text quality and orientation. Third, all other commands, namely the lower case letters and most nonalphanumeric characters, are interpreted by the controlling task and will terminate a cursor read. Thus, if any keystroke is entered that is not shown below or handled by the governing application program, cursor mode exits and the keystroke and cursor coordinates are reported.

Minimum match abbreviations are permitted for the cursor mode ":." command names. Multiple commands may be given on one line, delimited by semicolons.

The following upper-case commands are interpreted by the graphics system and may therefore be entered from the keyboard either in task mode or from "=gcur" (this is the same help panel you get from cursor mode by typing ":.help"):

A	draw and label the axes of current viewport
B	backup over last instruction in frame buffer
C	print the cursor position as it moves
D	draw a line by marking the endpoints

E	expand plot by setting window corners
F	set fast cursor (for HJKL)
H	step cursor left
J	step cursor down
K	step cursor up
L	step cursor right
M	move point under cursor to center of screen
P	zoom out (restore previous expansion)
R	redraw the screen
T	draw a text string
U	undo last frame buffer edit
V	set slow cursor (for HJKL)
W	select WCS at current position of cursor
X	zoom in, X only
Y	zoom in, Y only
Z	zoom in, both X and Y
<	set lower limit of plot to the cursor y value
>	set upper limit of plot to the cursor y value
\	escape next character
:	set cursor mode options
:!	send a command to the host system
=	short for ":.snap"
0	reset and redraw
1-9	roam

If the character : is typed while in cursor mode the alpha cursor will appear at the bottom of the screen, allowing a command line to be entered. Command lines which begin with a period, e.g., ":." are interpreted by the graphics system; any other command will terminate the cursor read. If not running an IRAF task which interprets that other command, cursor mode will be terminated and the cursor value reported.

:.axes[+-]	draw axes of viewport whenever screen is redrawn
:.case[+-]	enable case sensitivity for keystrokes
:.clear	clear alpha memory (e.g, this text)
:.cursor n	select cursor (0=normal,1=crosshair,2=lightpen)
:.gflush	flush plotter output
:.help	print help text for cursor mode
:.init	initialize the graphics system
:.markcur[+-]	mark cursor position after each cursor read
:.off [keys]	disable selected cursor mode keys
:.on [keys]	enable selected cursor mode keys
:.page[+-]	enable screen clear before printing help text
:.read file	fill frame buffer from a file
:.show	print cursor mode and graphics kernel status
:.snap [device]	make hardcopy of graphics display
:.txqual qual	set character generator quality (normal,l,m,h)
:.txset format	set text drawing parameters (size,up,hj,vj,etc)
:.xres=value	set X resolution (stdgraph only)
:.yres=value	set Y resolution (stdgraph only)
:.viewport x1 x2 y1 y2	set workstation viewport in world coordinates

:.write[!][+] file	save frame buffer in a spool file
:.zero	reset viewport and redraw frame

3. ADVANCED USAGE

3.1 THE FRAME BUFFER

The concept of the FRAME BUFFER is essential to an understanding of cursor mode. IRAF tasks output all graphics in the form of GKI metacode instructions. These instructions may be stored in a file if desired, or, if the task is run from the CL, they will usually be stored automatically in the frame buffer. This is a large storage area internal to the CL process, and is transparent to the user. What is important is that after producing a plot on the screen, all or part of the information in the plot is still present in the frame buffer. That means that it is possible to enter an interactive session with the plot, whether as a part of the task that produced the plot in the first place or after the task exits by typing "=gcur" from the CL.

If one wishes to recall the last plot after the task which created it has exited, and the screen has since been cleared, the plot will still be in the frame buffer and can be redrawn by entering cursor mode and typing 0 (the digit zero). If the desired plot was not the last one plotted, hence is no longer in the frame buffer, it can still be recalled if it was saved earlier in a metacode file on disk. The command ":.read fname" will refill the frame buffer from file "fname", and redraw the plot.

All graphics instructions output since the last time the device screen was cleared reside in the frame buffer unless there is an extremely large amount of information in the plot, in which case only the last part of the plot will be saved (the frame buffer dynamically sizes itself to fit the frame, but there is a fixed upper limit on its size of about 100 Kb).

3.2 FILLING AND WRITING THE FRAME BUFFER

The graphics system will automatically clear the frame buffer whenever the screen is cleared when plotting. For example, in a heavy interactive graphics session, the frame buffer will be filled and cleared many times, and at the end only the last screenful will be left in the frame buffer. When reading a metacode file containing several frames with ":.read", all frames will be plotted in sequence, but only the last one will remain in the buffer when the sequence finishes.

Some tasks have application-specific functions that append to, rather than overwrite the frame buffer. For example, the "j" function in

IMPLOT plots another line from the image. On the screen the previous data vectors are erased and the new ones drawn over. However, if you then do a zoom or a reset screen, you will see EACH of the sets of data vectors drawn in succession (some people unfairly consider this to be a bug, but actually it is a highly desirable feature which we are justifiably proud of).

The contents of the frame buffer may be written to a metacode file with ":.write file". By default the frame buffer is appended to the file if it already exists. If you wish to "clobber" an existing file, use ":.write! file". Also by default, the frame that is written is what you currently see on the screen, i.e., if you have zoomed in on a feature only what you see on the screen will be saved. To write the full frame (the one you would see if you first did a "0"), use ":.write+ file". To overwrite an existing metacode file in full-frame mode, use ":.write!+ file".

3.3 MOVING THE CURSOR AND MODIFYING THE DISPLAY AREA

A number of special keystrokes are recognized for interactive display control. These keystrokes may be used to redraw all or any portion of the spooled graphics; e.g., one may zoom in on a portion of the plot and then roam about on the plot at high magnification. Since the spooled graphics vectors often contain more information than can be displayed at normal magnification, zooming in on a feature may bring out additional detail (the maximum resolution is 32768 points in either axis). Increasing the magnification will increase the precision of a cursor read by the same factor.

If the graphics frame is a typical vector plot with drawn and labelled axes, magnifying a portion of the plot may cause the axes to be lost. If this is not what is desired a keystroke ("A") is provided to draw and label the axes of the displayed window. The axes will be overplotted on the current display and will not be saved in the frame buffer, hence they will be lost when the frame is redrawn. New axes may optionally be drawn every time the viewport changes after entry of the command ":.axes+". In cursor mode the viewport is the full display area of the output device, hence the tick mark labels of the drawn axes are drawn inside the viewport, on top of the data.

By default the cursor mode keystrokes are all upper case letters, reserving lower case for applications programs. The terminal shift lock key may be used to simplify typing in lengthy interactive cursor mode sessions. Most of the upper-case commands involve moving the graphics cursor and/or redisplaying a different part of the plot. Special keystrokes are provided for stepwise cursor motions to increase the speed of cursor setting on terminals that do not have fast cursor motions (e.g., the Retro-Graphics enhanced VT100). These keystrokes will only work if the terminal you are using permits positioning of the cursor under software control.

The commands H, J, K, and L (upper case!) move the cursor left, down, up, and right (as in the VI text editor and in Forth/Camera graphics). The step size of each cursor motion can change in one of three ways. "F" increases the step size by a factor over the current step size each time it is used; "V" decreases it similarly.

In practice the F/V speed keys are rarely used because the cursor positioning algorithm will automatically adjust the step size as you move the cursor. A large step size is used to cross the screen, then the step size is automatically decreased as you get close to the desired feature. Some practice is required to become adept at this, but soon it becomes natural and fast.

Arrow keys, thumbwheels, etc., if present on a keyboard, may also be used for cursor motions. However, moving the cursor this way does not automatically report the position to the graphics system, thus if the command "C" is given, you will not get a position report after each motion.

The numeric keypad of the terminal (if it has one) is used to roam about when the zoom factor is greater than one. A numeric key must be escaped to use it to exit cursor mode, i.e., if the applications program reading the cursor recognizes the digit characters as commands. The directional significance of the numeric keys in roam mode is obvious if the terminal has a keypad, and is illustrated below.

```
7   8   9       135 090 045

4   5   6       180 000 000

1   2   3      -135 -90 -45
```

Even if the terminal has a keypad, it may not be possible to use it for roam on some terminals. If the keypad does not work, the normal numeric keys at the top of the keyboard will, after a glance at the keypad to see which digit to use.

3.4 REPORTING AND MARKING THE CURSOR POSITION

If you wish to know the world (data) coordinates of a point on the screen, issue the command "C". Then, every time you move the cursor with HJKL, the world coordinates at each new position will be reported on the lower left line of the screen.

If the cursor mode option ":.markcur+" is set, the position of the cursor will be marked with a small plus sign when time cursor mode exits, returning the cursor position to the calling program. This is useful when marking the positions of a large number of objects, to keep track of the objects already marked. The cursor position will not be marked until cursor mode exits, i.e., no cursor mode command will cause the mark to be drawn. The mark cursor option remains in effect until you explitictly turn it off with ":.markcur-" or by typing GFLUSH. The marks are drawn in the frame buffer, hence they will survive zoom and roam or screen reset (they can be erased with repeated B commands if desired).

Some plots have more than one world coordinate system (WCS, the third value in the cursor value string). Suppose you are in cursor mode and the frame contains two separate plots, or there is only one plot but the lower x-axis is in Angstroms while the upper one is in inverse centimeters. By default the graphics system will automatically select the WCS (viewport) closest to the position of the cursor, returning a cursor position in that coordinate system. If this is not what is desired, move the cursor to a position that belongs unambiguously to one of the coordinate systems and type "W". Subsequent cursor reads will refer to the coordinate system you have specified, regardless of the position of the cursor on the screen. When the frame is cleared the WCS "lock" will be cleared as well.

3.5 ANNOTATING PLOTS

The "T" command will prompt you for a text string to be entered from the keyboard, followed by a RETURN. The text will appear on the screen (and get stored in the frame buffer), normally located with its lower left corner at the current cursor position. This command may be used in conjunction with the "D" command to draw a line from the text annotation to a feature of interest in the plot. Notice that the text size is constant in cursor mode regardless of the current magnification. In order that text entered with "T" will look as nearly the same as possible on a hardcopy snapshot as it does on the screen, you should set text quality to high.

Text attributes are controlled by two command options. Use ":.txqual" to set text quality to "normal" (the default), "low", "medium", or "high". Low-quality text plots the fastest, high-quality the slowest. On terminals with hardware text generation such as the Retro-Graphics Enhanced VT100, low-quality characters may always come out upright, even if the whole text string's up-vector is not at 90 degrees.

Low-quality text sizes are also fixed on most devices, so in a hardcopy snapshot of a plot the text will not necessarily look the same as it did on the screen (in particular it may overwrite data vectors). With low-quality text other options such as "font=italic" will not work on most terminals (although they may come out correctly on a hardcopy device). In general, set ":.txqual=h" if you are planning to get hardcopy output from a plot you are annotating. Changing the text quality only applies to text entered with "T" AFTER the change; you cannot automatically set all text to high quality after you have

entered it.

There are several ways to change the position of text relative to the cursor, its size, font, and orientation. Use ":.txset" to change the text drawing parameters as follows:

keyword	values	default
up	degrees counterclockwise, zero = +x	90
size	character size scale factor	1.0
path	left, right, up, down	right
hjustify	normal, center, left, right	left
vjustify	normal, center, top, bottom	bottom
font	roman, greek, italic, bold	roman
quality	normal, low, medium, high	normal
color	integers greater than one	1

The "up" keyword controls the orientation of the character and the whole text string. A text string oriented at +45 degrees to the horizontal, from left to right, would have "up=135".

Character sizes are all specified relative to a base size characteristic of each plotting device. The size is a linear magnification factor, so "size=2.0" results in a character with four times the area.

Path is relative to the up vector; a string of characters consecutively underneath each other with the normal upright orientation would have "up=90;path=down".

The justify parameters refer to the placement of the entire text string relative to the current cursor position. To center a text string horizontally over a spike in a plot, position the cursor to just above the spike and set "h=c;v=b".

Font and quality were discussed above. Setting the color will only have an effect on devices supporting it; if you have a color pen plotter, you must remember the current color setting, because there you cannot see it on the screen (":.show" will reveal it however).

If you make a mistake or don't like the appearance of the text you entered, all is not lost. Use the command "B" to back up over the last instruction and redraw (e.g. with "0") until you're ready to reenter the text. If you back up one instruction too far (you lose some of the data vectors for instance) just type "U" to undo the last frame buffer edit, i.e. the backup.

For example, to annotate a spectral line with "H-alpha", written sideways up the screen from the current position in italics:

```
:.txqual high
:.txset up=180;font=italic
T
```

```
        text: H-alpha
```

On the last line, cursor mode provided the "text: " prompt. The format could have been shortened to "u=180;f=i".

3.6 HARDCOPY SNAPSHOTS

There are two main ways to get a hardcopy of the frame buffer. To get a copy of what you see on the screen directly on a hardcopy plotter, simply use ":.snap plottername". When you do so, you are actually sending the output down a buffered stream. That is, you can do several ":.snap"'s before anything actually comes out on the plotter. This is because many plotters use several pages worth of blank paper before and after the actual plot. If you are planning to make a number of snapshots in succession, even if they are from different "=gcur" sessions, simply use ":.snap" for each one until you are done, then issue ":.gflush". You can also flush graphics output to a plotter from the CL using the Language Package task GFLUSH:

```
        cl> =gcur
        ...
        :.snap versatec
        ...
        :.snap versatec
        <RETURN>
        cl>
        cl> gflush
```

Alternatively, you can use ":.write mcodefile" as discussed above, appending as many different frames as you wish, then later from the CL, send the metacode file to a plotter with one of the graphics kernels:

```
        cl> implot
        ...                             (interactive session)
        :.write file1.mc
        <RETURN>
        cl> stdplot file1.mc

                or

        cl> calcomp file1.mc            (etc.)
```

3.7 ALTERNATE CURSOR INPUT

Any program which uses cursor input may be run noninteractively as well as in batch mode. For example, suppost the task has a cursor type

parameter called "coords". In normal interactive use a hardware cursor read will occur every time the program reads the value of the "coords" parameter. To run the program in batch mode we must first prepare a list of cursor values in a text file, e.g., with the RGCURSOR or RIMCURSOR tasks in the LISTS package. We then run the task assigning the name of the cursor list file to the parameter "coords". For example, to run the APPHOT task in batch, with the cursor list in the file "starlist":

```
        cl> apphot arg1 arg2 ... argN coords=starlist &
```

The program will then read successive cursor values from the starlist file, not knowing that the cursor values are coming from a text file rather than from actual cursor reads.

A second mechanism is available for redirecting cursor input to the terminal. This is most useful when working from a terminal that does not have graphics, or when debugging software. To work this way one must first set the value of the environment variable STDGCUR (for the graphics cursor) or STDIMCUR (for the image cursor). Set the value to "text" to direct cursor reads to the terminal, e.g.:

```
        cl> set stdgcur = text
```

The cursor value will then be a line of text read from the user terminal. In this mode the user enters at least two of the fields defining a cursor value. Missing fields are assigned the value zero (the user presumably will know that the program does not use the extra fields).

```
        cl> = gcur
        gcur: 345.33 23.22 1 c
        345.33 23.22 1 c
        cl>
```

An example of a cursor read request entered interactively by the user, taking input from the terminal and sending output to the terminal, is shown above (the CL typed the "gcur: " query and the user entered the remainder of that line). If the cursor device were "stdgraph" a real cursor read would occur and the equivalent interaction might appear as shown below. The cursor position is returned in world coordinates, where the world coordinate system was defined by the last plot output to the device. For an imaging device the world coordinates will typically be the pixel coordinates of the image section being displayed.

```
        cl> = gcur
        345.33 23.22 1 c
        cl>
```

Redirecting cursor input to the terminal is useful when working from a nongraphics workstation and when debugging programs. ASCII cursor queries are the only type supported when running an IRAF program outside the CL. Cursor input may also be taken from a list file by assigning a filename to a cursor parameter, i.e., by assigning a list file to a list structured parameter and overriding query mode:

```
cl> gcur = filename
cl> = gcur
345.33 23.22 1 c
cl>
```

3.8 EXAMINING THE STATUS OF THE GRAPHICS SYSTEM

The command ":.show" writes out a page of information concerning the state of the graphics system. This is an example of such a status report:

```
    Cursor Mode Parameters:

        case    = YES
        markcur = YES
        page    = YES
        axes    = NO
        view    = full screen
        keys    = ABCDEFGHIJKLMNOPQRSTUVWXYZ<>0123456789?:
                ->ABCDEFGHIJKLMNOPQRSTUVWXYZ<>0123456789?:


    Graphics Kernel Status:

        STDGRAPH: kernel=cl, device=vt640
            memory=9472 (8192fb+256sb+1024fio), frame=1114+0 words
            spool=yes, nopen=0, pid=0, in=0, out=0, redir=-6, wcs=0
            text size = 1., up=90, path=right, hj=left, vj=bottom, color=1

        STDIMAGE:       disconnected
        STDPLOT:        disconnected
```

The cursor mode parameters report the current values of the ":." command options; these options are in effect for all of three the standard graphics streams, i.e., STDGRAPH (the graphics terminal), STDIMAGE (the image display), and STDPLOT (batch plotters).

The graphics kernel status reports the status of each of the three graphics streams. These streams are independent and in principle any graphics device may be connected to any stream. The KERNEL field gives the name of the kernel connected to that stream, if any. The value "cl" refers to the STDGRAPH kernel, which is built into the CL, and which can only talk to graphics terminals. Any other value is the filename of an external graphics kernel, running as a subprocess of the CL process. The DEVICE field gives the name of the device named in the last "open workstation" command on that stream. This is the device the

stream is currently writing plots to.

The significance of the remaining kernel status fields is described below.

```
        memory          - total memory used, chars
        fb              - size of primary frame buffer, chars
        sb              - size of scratch frame buffer (used by A)
        fio             - size of the FIO buffer for the stream
        frame           - amount of data in the frame + data in SB

        spool           - enable spooling of graphics in frame buffer?
        nopen           - open count (should be zero)
        pid             - process id of kernel subprocess
        in              - fd of process in, if subkernel
        out             - fd of process out, if subkernel
        redir           - redirection information for pseudofile i/o
        wcs             - current WCS, zero if not locked with W

        text size       - current text size relative to device’s base size
        up              - text up vector
        path            - text character drawing path
        hj              - horizontal justification
        vj              - vertical justification
        color           - index of current color attribute
```

This status report reflects only the information known to the CL. The graphics subkernels, which are subprocesses of the CL, may themselves have subprocesses, sometimes on different nodes in the local network.

3.9 INITIALIZING THE GRAPHICS SYSTEM

The graphics system can normally be initialized by typing GFLUSH. This will clear the frame buffer and disconnect all kernels, freeing memory and file descriptors, and reducing the subprocess count. Shutting down a graphics subkernel automatically flushes any buffered graphics output. The CL automatically calls GFLUSH during logout to shutdown the graphics system in an orderly fashion.

BUGS

Despite the fact that the CL has graphics and image cursor access capabilities, there is no guarantee that one can access the cursor on a particular device. A GRAPHCAP entry for the device is also required, as is a graphics kernel if the device is not a conventional graphics terminal (e.g., an image display). If all of these pieces are not in place, the system will abort the cursor read, complaining that it cannot find a termcap or graphcap entry for the device, or that it cannot open a connected subprocess (the subkernel).

SEE ALSO
The GIO Reference Manual

NAME
declarations -- parameter and variable declarations

SYNTAX

```
vartype [*]varname[index_list] [= init_value] [{options}] [, ...]

  or

vartype [*]varname[index_list] [{init_value [, options]}] [, ...]
```

ELEMENTS

vartype
One of the legal variable types, i.e.:

int, bool, char, real, gcur, imcur, struct, file

varname
The name of the variable or parameter. The name must begin with an alphabetic character or '_' and should be fewer than 64 characters in length. If the name is preceded by a '*', then the variable is 'list-directed', meaning that a new value is taken from a list each time the parameter is read.

index_list
The index_list consists of a series of ranges enclosed in square brackets. A range may be a single integer in which case the range is from 1 to that integer, or two integers separated by a colon. The second integer must be larger than the first. Ranges are separated by commas. In the special case that no ranges are specified by the user, the variable is assumed to be a one-dimensional array with a range from 1 to the number of elements in the initialization list.

init_value
The initialization value is a single value for scalar parameters but may be a list for array. A repetition count may be specified in the form

rep_count (value)

which is equivalent to value repeated the rep_count times. The values in the initialization list are separated by commas.

options
Options define certain characteristics of the variables. Each options has the form opt_name=value where value is a constant. The current options are:

mode
Determines whether the parameter is queried for and whether it is learned after task execution. The default mode for parameters declared in the argument list of a CL procedure is "a", and "h" otherwise.

min The minimum allowable value for the parameter. If omitted, no min checking is performed.

max The maximum allowable value for the parameter. If omitted, no max checking is performed.

prompt
The prompt to be used when the parameter is queried for.

enum
The set of allowable string values for a string valued parameter. The character '|' delimits successive enumerated strings.

filetype
For a FILE type parameter, a string containing characters giving file characteristics to be checked for when the file parameter is used.

```
r       file exists and is readable
w       file exists and is is writable
n       file does not exist
b       file is a binary file
t       file is a text file
```

length
For a string type parameter, the number of characters of storage to allocate for the string. If the actual length of a string later exceeds the allocated value the string will be silently truncated.

Note that all string constants in an options list must be enclosed in quotes.

DESCRIPTION

Declaration statements are used for inline declaration of parameters and local variables. A declaration after the begin statement of a procedure script is a declaration of a local variable, but any other declaration defines a parameter. Parameters are generally saved between invocations of a script while local variables are not.

Parameter and variable declarations should always precede executable statements with a script. Certain functions are legal before declarations, but this depends upon certain hidden aspects of declarations which are not obvious to the user.

DECLARATIONS (Feb86) language DECLARATIONS (Feb86)

EXAMPLES

```
real    x
int     ii=32
int     y {min=0, max=14}
char    z="abc" {enum="abc|def|ghi", mode="q"}

bool    isotest {YES, mode="ql",
            prompt="Do you want to test for isotropy?"}

int     ii=1 {min=0,max=10, prompt="Number of images", mode="h"}
file    infile="testfile" {filetype="r"}
struct  line {length=80, mode="h"}

real    array[10]
int     iarray[15]=1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 {min=0, max=100}
int     jarray[15] { 5(0), 5(2), 5(4), min=0, max=400}
char    carray[5]= 5("Junk")
bool    flags[4,-3:3] = 28(NO) {mode="h", prompt="Value set"}
file    inp_files[3]= "fil1.inp", "fil2.inp", "fil3.inp"

int     karray[3]=1     # (note second and third elements are undefined)
struct  *list="inputfile.list" {mode="q"}
int     *ilist="infile.inp" {mode="h", min=0, max=100}
```

BUGS

Options are only permitted for parameters, not local variables.
The filetype options are recognized but are not implemented internally.

SEE ALSO

parameters, procedure, begin

NAME
switch -- switch case statement

SYNTAX
```
switch (expr) {
case val1 [, val1,...]:
    statements
case val3 [, val3,...]:
    statements
        (etc.)
default:
    statements
}
```

ELEMENTS

expr
An integer-valued expression tested before entry into the switch block.

valN
Integer valued constants used to match expression.

statements
Simple or compound statements to be executed when the appropriate case or default block is selected.

DESCRIPTION
The SWITCH statement provides a multiway branch capability. The switch expression is evaluated and control branches to the matching CASE block. If there is no match the DEFAULT block, if present, receives control. If no DEFAULT block is present, the switch is skipped.

Each CASE statement consists of a list of values defining the case, and an executable statement (possibly compound) to be executed if the case is selected by the switch. Execution will continue until the next case is reached, at which time a branch out of the SWITCH statement occurs. Note this difference from the C switch case, where an explicit BREAK statement is required to exit a switch. If a BREAK is used in a CL switch, it will act upon the loop statement containing the switch, not the switch itself.

Note that both the switch expression and the case constants may be integers, or single characters which are evaluated to their ASCII equivalents.

The DEFAULT statement must be the last statement in the switch block.

DEFAULT (Feb86) language DEFAULT (Feb86)

EXAMPLES
1. Multiple cases, no default case.

```
switch (opcode) {
case 1:
    task1 (args)
case 2:
    task2 (args)
case 5:
    task5 (args)
}
```

2. Multiple values in a case.

```
switch (digit) {
case '1','2','3','4','5','6','7':
    n = n * 8 + digit - '0'
default:
    error (1, "invalid number")
}
```

BUGS
Only integer values are allowed (no strings). The case values must be constants; ranges are not permitted.

SEE ALSO
if else, goto

NAME
    defpac -- test if the named package is defined
    deftask -- test if the named task is defined
    defpar -- test if the named parameter is defined

USAGE
    defpac (pacname)
    deftask (taskname)
    defpar (param)

PARAMETERS

pacname
An IRAF package name.

taskname
An IRAF taskname. It may be specified as "taskname" or as "packagename.taskname".

param
An IRAF parameter name. It may be specified as "paramname", "taskname.paramname" or "packagename.taskname.paramname".

DESCRIPTION
These routines return a boolean value indicating whether the relevant parameter, task or package has been defined. A task becomes defined when the package to which it belongs is "loaded" by entering the name of the package as a command, or whenever a TASK declaration is input to the CL. A parameter becomes defined when the task to which it belongs is defined; the task need not be currently executing for its parameters to be defined. When a package is exited, e.g., after entry of the BYE command, all the task and parameter declarations for the package are discarded.

EXAMPLES
1. Test if a task exists.

```
cl> if (deftask ("system.page"))
>>>     print ("task page exists")
>>> else
>>>     print ("task page not found")
task page exists
cl>
```

2. Add the value of the named parameter into a sum, but only if the parameter exists (the example is for a script).

```
sum = 0
```



```
for (i=0;  i <= 10;  i+=1) {
    parname = "data" // i
    if (defpar (parname)
        sum += parname
}
```

SEE ALSO
    package, task, redefine, lparam

NAME
    defpac -- test if the named package is defined
    deftask -- test if the named task is defined
    defpar -- test if the named parameter is defined

USAGE
    defpac (pacname)
    deftask (taskname)
    defpar (param)

PARAMETERS

pacname
An IRAF package name.

taskname
An IRAF taskname. It may be specified as "taskname" or as "packagename.taskname".

param
An IRAF parameter name. It may be specified as "paramname", "taskname.paramname" or "packagename.taskname.paramname".

DESCRIPTION
These routines return a boolean value indicating whether the relevant parameter, task or package has been defined. A task becomes defined when the package to which it belongs is "loaded" by entering the name of the package as a command, or whenever a TASK declaration is input to the CL. A parameter becomes defined when the task to which it belongs is defined; the task need not be currently executing for its parameters to be defined. When a package is exited, e.g., after entry of the BYE command, all the task and parameter declarations for the package are discarded.

EXAMPLES
1. Test if a task exists.

```
cl> if (deftask ("system.page"))
>>>     print ("task page exists")
>>> else
>>>     print ("task page not found")
task page exists
cl>
```

2. Add the value of the named parameter into a sum, but only if the parameter exists (the example is for a script).

```
sum = 0
```



```
for (i=0;  i <= 10;  i+=1) {
    parname = "data" // i
    if (defpar (parname)
        sum += parname
}
```

SEE ALSO
    package, task, redefine, lparam

NAME
    defpac -- test if the named package is defined
    deftask -- test if the named task is defined
    defpar -- test if the named parameter is defined

USAGE
    defpac (pacname)
    deftask (taskname)
    defpar (param)

PARAMETERS

    pacname
        An IRAF package name.

    taskname
        An IRAF taskname. It may be specified as "taskname" or as "packagename.taskname".

    param
        An IRAF parameter name. It may be specified as "paramname", "taskname.paramname" or "packagename.taskname.paramname".

DESCRIPTION
    These routines return a boolean value indicating whether the relevant parameter, task or package has been defined. A task becomes defined when the package to which it belongs is "loaded" by entering the name of the package as a command, or whenever a TASK declaration is input to the CL. A parameter becomes defined when the task to which it belongs is defined; the task need not be currently executing for its parameters to be defined. When a package is exited, e.g., after entry of the BYE command, all the task and parameter declarations for the package are discarded.

EXAMPLES
    1. Test if a task exists.

```
cl> if (deftask ("system.page"))
>>>     print ("task page exists")
>>> else
>>>     print ("task page not found")
task page exists
cl>
```

    2. Add the value of the named parameter into a sum, but only if the parameter exists (the example is for a script).

```
sum = 0
```



```
for (i=0;  i <= 10;  i+=1) {
    parname = "data" // i
    if (defpar (parname)
        sum += parname
}
```

SEE ALSO
    package, task, redefine, lparam

NAME
dparam -- dump the parameters of a pset as a series of assignments

USAGE
dparam pset [pset ...]

PARAMETERS

pset
The name of the parameter set to be listed.

DESCRIPTION
DPARAM lists one or more parameter sets. Psets are specified either by the name of the task with which the pset is associated, or by filename (pset files have the ".par" extension). If a file type pset is listed the extension must be included, since it is the presence or absence of the filename extension which DPARAM uses to distinguish between task-psets and named (file) psets.

Each parameter is listed on a single line with the following format. The list of assignments is terminated by the string "# EOF" so that programs reading the list from a stream can easily distinguish the end of the variable length list of parameters.

```
task.param = value
```

Here "task.param" is the name of the parameter, and "value" is the current value of the parameter. The assignment is skipped if the value is undefined. There is no way to distinguish between hidden parameters and query parameters.

The output from DPARAM is often used as input to programs, whereas the output from LPARAM is formatted in a way which makes it easier for humans to read. For example, the output from DPARAM may be redirected into a file and used on the IRAF main command line to set the task’s parameters, when debugging a task standalone.

EXAMPLES
1. List the parameter for the task DELETE. Note that the positional parameters are listed first, in the order in which they must be specified on the command line, followed by the hidden parameters.

```
cl> dparam delete
delete.files = "temp"
delete.go_ahead = yes
delete.verify = no
delete.default_action = yes
delete.allversions = yes
```



```
delete.subfiles = yes
delete.mode = "ql"
# EOF
```

2. List the contents of the file pset "delete.par". Named psets such as this are most commonly produced using the ":W FILENAME" colon command in EPARAM, e.g., to prepare several different versions of the parameter set for a task.

```
cl> dparam delete.par
```

BUGS
You cannot list the parameters of a task that does not have a parameter file (e.g., all builtin tasks).

SEE ALSO
eparam, lparam, cache

NAME

edit -- edit a text file

USAGE

edit files [files...]

PARAMETERS

files
The file or files to be edited.

DESCRIPTION

The EDIT task invokes a host system editor to edit the named file or files. The editor to be used is determined by the value of the CL environment variable EDITOR. Filename mapping is applied to the FILES argument strings to convert virtual filenames into host system filenames. File templates are not supported, unless the host system editor supports them.

The EDT, EMACS, and VI editors are currently supported. Each editor interface is controlled by an EDCAP table file in the logical directory "dev$"; these files are also used by the EHISTORY and EPARAM screen editors. For example, the file "dev$edt.ed" is required to run the EDT editor. The EDITOR_CMD field of the EDCAP file defines the command to be send to the host system to run the editor; this is not necessarily the same as the name of the editor. Support for additional editors is easily added by adding new EDCAP files.

EXAMPLES

1. Edit the login.cl file.

```
cl> edit home$login.cl
```

2. Edit the file "temp" in the current directory.

```
cl> edit temp
```

3. On a UNIX system, edit all the ".x" files in the current directory. Filename templates cannot be used with the editor unless the editor itself, or the host system, expands the filename template.

```
cl> edit *.x
```


BUGS

The EOF control character is set in the edcap file for the editor language in use, e.g., "dev$vi.ed" for the VI editor. The value in this file may differ from that used on the local system; if this is the case, the system installer should edit the file and change the value of the parameter EXIT_UPDATE.

The control sequences for the keyboard arrow keys are also defined in the ".ed" edcap file; TERMCAP should be used instead.

SEE ALSO

ehistory, eparam

NAME
ehistory -- edit and re-execute previous commands

USAGE
ehistory (or just "e")

PARAMETERS
None.

DESCRIPTION
The EHISTORY command calls up a screen editor to edit previously executed commands, executing the edited command when return is typed. Interrupt (e.g., <ctrl/c> may be used to escape from the editor at any time. The type of editor commands recognized is determined by the value of the CL environment variable EDITOR, which may currently be set to "edt", "emacs", or "vi".

After the EHISTORY command is entered, the previous command is displayed at the bottom of the terminal. If the previous command was a compound statement, or if it extended over more than one line, all the lines of the command will be displayed. To reach a different command simply enter the appropriate cursor movement keys for the editor type being used. When the cursor attempts to move above the current command the previous command will be displayed. Similarly when it attempts to move below, the next command will appear. Hitting the return key will execute the command currently being edited.

The CL parameter "ehinit" may be used to set the following options:

[no]standout
Controls whether the command to be edited is displayed in reverse or normal video.

eol The editor is entered with the cursor positioned to be end of the command line.

bol The editor is entered with the cursor positioned to be beginning of the command line.

[no]verify
If VERIFY is specified, EHISTORY will be automatically entered when history commands are entered to recall and execute previous commands. If NOVERIFY is specified, the commands are recalled and immediately executed.

EHISTORY (Jun86) language EHISTORY (Jun86)

EXAMPLES
1. Set no standout and verify modes.

```
cl> ehinit = "nostandout verify".
```

2. Recall the last "xc" command from the history list and edit it. If VERIFY were not enabled the command would simply be repeated.

```
cl> ^xc
```

BUGS
The command editor really only works well for single line commands; multiline command blocks are not easily edited at present. VI is poorly emulated at present since only control code editor commands are possible.

SEE ALSO
eparam

NAME
    if else -- conditional execution of a statement

SYNTAX
    IF (expression) statement1 [ELSE statement2]

ELEMENTS

    expression
        A boolean valued expression.

    statement1, statement2
        The statements to be executed (possibly compound, i.e., enclosed in curly braces).

DESCRIPTION
    The IF statement is used to execute a statement only if the specified condition is true. An optional ELSE clause may be given to execute a different statement if the condition is false.

EXAMPLES
    1. Add X to Y only if X is less than Y.

```
        if (x < y)
            y += x
```

    2. If X is less than 10 print "small", else print "big".

```
        if (x < 10)
            print ("small")
        else
            print ("big")
```

    3. The ELSE IF construct.

```
        if (str == "+")
            val += x
        else if (str == "-")
            val -= x
        else if
            ...
```

    4. Nesting, use of braces.

```
        if (i > 0) {
            if (i < 10) {
                print ("0")
                sum = sum * 10 + i
```



```
            } else
                print (" ")
        }
```

SEE ALSO
    for, case, break, next

NAME
envget -- get the string value of an environment variable

USAGE
envget varname

PARAMETERS

varname
    The environment variable whose value is to be returned.

DESCRIPTION
ENVGET returns the string value of the named environment variable. The user is prompted for the value if the variable has not yet been defined.

EXAMPLES
1. Construct a filename using the value of the environment variable "editor", and page the file thus named.

```
cl> page ("dev$" // envget ("editor") // ".ed")
```

2. Compute and print the center line on the terminal screen.

```
cl> = ((int (envget ("ttynlines")) + 1) / 2)
```

SEE ALSO
set, show

NAME
eparam -- edit a task's parameters

USAGE
eparam task [task ...]

PARAMETERS

task
    The name of the task whose parameter set is to be edited.

DESCRIPTION
The EPARAM command calls up an interactive screen editor to edit the parameters of the named task or tasks. The syntax of the page editor is controlled by the environment variable ‘editor’ which may have the values "edt", "emacs", or "vi". The user may also customize the editor by copying the associated "dev$*.ed" file to their home directory, and editing the file.

The CL parameter "epinit" may be used to set the following options:

[no]standout
    Enables or disables use of standout mode (reverse video) in the display.

[no]showall
    Controls whether or not hidden parameters are displayed and edited.

The EPARAM task may be used to edit either ordinary task parameter sets, or named parameter files. The presence or absence of a .PAR filename extension is used to determine whether an operand is a taskname or a filename. For example,

```
cl> eparam skypars.par
```

will edit the parameter FILE SKYPARS.PAR in the current directory, whereas

```
cl> eparam skypars
```

will edit the parameter set for the pset-task SKYPARS. Lastly, since SPYPARS is a pset-task, we could just type

```
cl> skypars
```

to edit or review the contents of the pset.

The parameter file SKYPARS.PAR in the above example would probably be created using the new colon-command extensions to eparam. The original eparam supported only single keystroke editing commands. The new colon commands are used to enter command lines of arbitrary length to be processed by eparam.

A colon command is entered by typing the colon character (':') while the cursor is positioned to the starting column of any value field of the parameter set being edited. The colon character is not recognized as a special character beyond column one, e.g., when entering the string value of a parameter. When colon command mode is entered, the colon character will be echoed at the start of the bottom line on the screen, and the cursor will move to the character following the colon, waiting for the command to be entered. The command is read in raw mode, but the usual delete, <ctrl/c>, <ctrl/u>, etc. sequences are recognized.

The following eparam colon commands are currently supported. All commands are carefully error checked before being executed to avoid having eparam abort with a stack trace. An illegal operation causes colon command entry mode to be exited, leaving an error message on the command entry line. All commands which cause editing of the current pset to terminate may include the ! character to avoid updating the current pset before reading in the new one or exiting eparam. The default is to update the current pset. In all cases, PSET may be either the name of a task or the name of a parameter file. Parameter files are always indicated by a .PAR extension, even though the actual file may be a .CL file: only .PAR files will be written, although either type of file may be read.

:e[!] [pset]
Edit a new pset. If PSET is omitted and the cursor was positioned to a pset parameter when the colon command was entered then eparam descends into the referenced pset; when editing of the sub-pset is complete eparam returns to editing the higher level pset at the point at which the ':E' command was entered. If a pset is named the editor context is switched to the new pset, updating the current pset first unless the ':E!' command was given.

:q[!]
Exit eparam for the current pset; equivalent to a <ctrl/z>. The variant ':Q!' causes eparam to be exited without updating the current pset. Entering this command when editing a sub-pset causes an exit to the higher level pset. To abort eparam entirely without updating anything, <ctrl/c> should be used.

:r[!] [pset]
Read in a new pset. If the command is ':R', an error message is printed. If the command is ':R!' the pset currently being edited is reread, cancelling any modifications made since the last update. If a pset is specified the contents of the named pset are merged into the current pset, i.e., the named pset is loaded into the current pset, overwriting the contents of the current pset. The command ':R PFILE.PAR' is commonly used to load a pset formerly saved in a user file with ':W PFILE.PAR' into the UPARM version of the parameter set for a task.

:w[!] pset
Write or update a pset. If PSET is omitted the pset currently being edited is updated on disk. If PSET is given it should normally be the name of a parameter file to be written. If the file exists an error message will be printed unless the command ':W! PFILE.PAR' is given to force the file to be overwritten.

:g[o][!]
Run the task. Eparam exits, updating the pset and running the task whose pset was being edited. This is implemented by pushing a command back into the input stream of the task which called eparam, hence if eparam was called in a script or with other commands on the same line, execution may be delayed until these other commands have been edited. The feature works as expected when used interactively. Since the run command is pushed back into the command input stream it will appear in the history record and in any log files.

To get out of colon command mode without doing anything, simply type delete until the colon prompt is deleted and the cursor returns to the parameter it was positioned to when colon command entry mode was entered.

EXAMPLES

1. Set standout mode and disable the editing of hidden parameters (leaving only the positional parameters).

```
cl> epinit = "standout noshowall"
```

2. Edit the parameters for the DELETE task.

```
cl> ep delete
```

SEE ALSO

lparam, ehistory

NAME
    error -- abort a task

USAGE
    error errcode errmsg

PARAMETERS

    errcode
        An integer code identifying the error (not used at present in the CL since error handlers are not supported).

    errmsg
        A string describing the error.

DESCRIPTION
    ERROR may be used to force an error exit from a script. The error message will be displayed, and control will return to the most recent interactive cl.

EXAMPLES
    1. Abort the current task if there is an attempt to compute a negative square root.

```
if (x < 0)
    error (1, "sqrt of a negative number (x=" // x // ")")
else
    y = sqrt (x)
```

BUGS
    There is currently no way to post an error handler to receive control if ERROR is called.

SEE ALSO
    cl, bye, logout

NAME
    flprcache -- flush the process cache

USAGE
    flprcache process

PARAMETERS

    process
        Either the task number as printed by PRCACHE, or the name of one of the tasks in the process. If no process is named, all processes are flushed from the cache (unless they are locked in the cache).

DESCRIPTION
    When an executable task is first run, the CL spawns the associated executable file as a subprocess and then runs the task. When the task completes the process does not exit, rather it remains connected to the CL as a subprocess, but becomes idle waiting for another command from the CL. The set of such idle processes forms what is referred to as the CL "process cache". The purpose of the process cache is to minimize the overhead required to run a task; the first time a task is called response is slow since the process has to be executed, but thereafter response is fast provided the process remains in the cache.

    The FLPRCACHE command flushes the process cache, terminating the connected subprocesses therein. If an argument is specified only the specific cache slot is cleared, otherwise all cache slots are flushed. Processes which have been "locked" in the cache with PRCACHE are not flushed unless explicitly named.

EXAMPLES
    1. Run PRCACHE to get the process slot number, then flush the process by slot number.

```
cl> flpr 5
```

    2. Flush all idle processes which are not locked in the cache.

```
cl> flpr
```

    3. Flush the "x_system.e" process by naming the "directory" task, which is contained in that process. Lock a fresh copy of the process in the cache. This initializes the process, and may be necessary if a system task is interrupted at the wrong time.

```
cl> flpr dir; prc dir
```

BUGS

In some circumstances the CL may believe that a process in the process cache is running when this is not the case. The CL will not attempt to communicate with a running process, and will be unable to kill the process. If this happens the CL will hang up during logout and will have to be interrupted, causing a panic abort (this is harmess since the CL is then restarted). The user may eventually be required to kill the sub-process using operating system facilities.

SEE ALSO

prcache

NAME

for -- FOR statement

SYNTAX

for ([assign1] ; [bool_expr] ; [assign2]) statement

ELEMENTS

assign1
An assignment used to initialize the FOR loop.

bool_expr
A boolean valued expression tested before each iteration.

assign2
An assignment executed after each iteration of the loop.

statement
A statement (possibly compound, i.e., enclosed in curly brackets) to be executed in each iteration of the loop.

DESCRIPTION

The FOR statement provides a looping mechanism similar to the C-language for loop. ASSIGN1 and ASSIGN2 are assignment statements using one of the operators '=', '+=', '-=', '/=', '*='. Any of the elements of the FOR loop may be omitted, except the parenthesis and colon field delimiters.

EXAMPLES

1. For I equals zero to 10 in steps of 2, increment TOTAL by the value of array element I.

```
for (i=0;  i <= 10;  i += 2)
    total += array[i]
```

2. Print the first eight powers of two.

```
j = 1
for (i=1;  i <= 8;  i += 1) {
     print (i, j)
     j *= 2
}
```

BUGS

A simple assignment of the form i++ will not work. Only one assigment statement is permitted in the first and third fields.

SEE ALSO
    while, case, break, next

NAME
    fprint -- print to a parameter
    print -- print to the standard output

USAGE
```
print  expr [expr ...]
fprint param expr [expr ...]
```

PARAMETERS

    expr
        Any expression, the string value of which is to be printed.

    param
        FPRINT will deposit the output string in the value field of this parameter.

DESCRIPTION

The PRINT and FPRINT commands format a line of text and write it to either the standard output or in the case of FPRINT, the p_value field of the named parameter. The output is free format although spaces may be specifically inserted (as quoted string constants) to make the output easier to read. One space is automatically inserted after each numeric argument; this can be defeated by coercing the argument to a string with the STR intrinsic function. A newline is automatically output at the end of the output line. I/O redirection may be used with PRINT to write to a file.

Compute mode (a parenthesized argument list) is recommended for this task to avoid surprises.

EXAMPLES

1. Print the name of the current terminal.

```
cl> print ("terminal = ", envget ("terminal"))
```

2. Output a blank line on the standard output, e.g., in a script.

```
print ("")
```

3. Format a command and send it to the host system. In this example, "fname" is a string valued parameter.

```
cl> print ("!dir/full ", fname) | cl
```

4. Write to a file.

```
for (x=1.;  x < 1E5;  x *= 10)
```

```
print ("the sqrt of ", x, "is ", sqrt(x), >> "output")
```

BUGS

The FPRINT task is not very useful since the same thing can be accomplished by string concatenation and assignment.

SEE ALSO

scan, fscan

NAME

scan -- read parameters from standard input
fscan -- read parameters from file, or another parameter
nscan -- get number of parameters scanned

USAGE

```
scan  (p1, p2, p3 ... pn)
fscan (param, p1, p2, p3, ... pn)

n = nscan()
```

PARAMETERS

pN The name of an output parameter, to receive a scanned value.

param
The name of the input parameter whose VALUE is to be scanned to produce the output values.

DESCRIPTION

SCAN and FSCAN permit the user to read in values from the standard input, a file, or a parameter and assign them to the listed parameters. FSCAN may also be used to read a string already in core. It is useful to consider these functions as performing two disjoint actions: acquiring a string, where SCAN and FSCAN differ; and parsing the string, where they are identical.

SCAN acquires its string by reading exactly one line from the standard input. The action of FSCAN depends on PARAM. If PARAM is a string, or a struct, then the string is simply the value of PARAM. If, however, PARAM is a list-directed struct, a call to FSCAN will get the next line from the file pointed to by PARAM. The file can be rewound by assigning a file name to PARAM. If either scan or fscan reach an EOF, they return with the value EOF and do not change any parameters.

Once the string has been acquired it is parsed into segments delimited by spaces or tabs. Scan and fscan do not recognize quoted strings, nor do they view ',' as a delimiter. Each token is then assigned in turn to p1 through pn. If there are too many tokens they are discarded, if there are too few, the corresponding parameters are not affected by the call. Any conversion error terminates the scan, but parameters already scanned retain their new values. An assignment to a struct terminates the scan because the entire unscanned portion of the string is assigned to the struct. Thus any struct should be the last parameter in a scan or fscan call.

Scan and fscan are intrinsic functions returning either EOF if end of file on the input list is sensed, or the number of parameters successfully scanned. The function NSCAN also returns the number of parameters successfully scanned in the last call to scan or fscan.

EXAMPLES

1. Print a list of radii, given a list of coordinates.

```
list = coords
while (fscan (list, x, y) != EOF)
    print (sqrt (x**2 + y**2))
```

BUGS

The syntax of scan and fscan is peculiar, in that they are the only functions where parameters are effectively passed by reference rather than by value. Thus p1, ... pn must be parameters whereas in similar contexts an arbitrary expression can be used wherever a parameter can.

SEE ALSO

string

NAME

gflush -- flush any buffered graphics output

USAGE

gflush

PARAMETERS

None.

DESCRIPTION

Output to graphics plotter devices is normally buffered and then disposed of to the plotter as a larger job, to increase the efficiency of the graphics system. The GFLUSH task disposes of any buffered graphics output and also initializes the graphics subsystem. The cursor mode frame buffer is cleared, any connected graphics subkernels are disconnected, and the memory and file descriptors used by the graphics subsystem are freed. A GFLUSH occurs automatically upon logout from the CL.

The number of frames (plots) the graphics system will buffer for a device is controlled by the MF (multi-frame) parameter in the GRAPHCAP entry for the device. When the multi-frame count is reached the buffered output is automatically disposed of to the device.

EXAMPLES

1. Flush any graphics output and initialize the graphics system.

```
cl> gflush
```

SEE ALSO

cursor, stdplot

NAME
    goto -- branch to a label

USAGE
```
goto label
  .
  .
  .
label: statement
```

PARAMETERS

label
    The destination label. Label names have the same syntax as variable names and can duplicate the names of existing variables.

statement
    The statement executed after the goto statement. It may be any executable statement.

DESCRIPTION
    The GOTO statement interrupts the normal flow of program execution by transferring control to the statement following the label. It may also be used to exit from nested loops where the break statement is not adequate.

EXAMPLES
    1. The most common use of the GOTO statement is to branch to an error handler if an abnormal condition is detected.

```
begin
        for (i=1;  i <= 100;  i += 1)
            for (j=1;  j <= 100;  j += 1)
                for (k=1;  k <= 100;  k += 1)
                    if (pixel[i,j,k] < 0)
                        goto err
                    else
                        total += pixel[i,j,k]

        print ("total = ", total)
        return
err:
        print ("Invalid pixel value at ",i,j,k)
end
```


GOTO (Feb86) language GOTO (Feb86)

BUGS
    No checking is done to see if a jump is made into a loop.

SEE ALSO
    break, next

NAME
    hidetask -- hide a task from the user

USAGE
    hidetask task [task ...]

PARAMETERS

    task
        The name of a task to be made hidden.

DESCRIPTION
    If a task is only to be called from other tasks, and is not normally invoked directly by the user, then it may be useful to ‘hide’ the task, i.e., omit it from the list of tasks listed in the "?" and "??" commands. The HIDETASK command performs this function.

EXAMPLES
    1. Define the task "_rew" and hide it from the user. The purpose of the leading underscore (not required) is to ensure that the user does not accidentally run the task.

```
cl> task $_rew = "home$rew.e"
cl> hide _rew
```

    2. Display the contents of the LANGUAGE package, including all hidden tasks (the _ does the trick).

```
cl> ?_ lan
    language:
      ?            chdir        defpar       history      radix
      ??           cl           deftask      jobs         redefine
      _allocate    clbye        edit         keep         scan
      _curpack     clear        ehistory     kill         service
      _deallocate  clpackage    envget       language     set
      _devstatus   d_f          eparam       logout       show
      access       d_l          error        lparam       sleep
      back         d_off        flprcache    mktemp       task
      beep         d_on         fprint       osfn         time
      bye          d_p          fscan        package      unlearn
      cache        d_t          gflush       prcache      update
      cd           defpac       hidetask     print        wait
```

SEE ALSO

NAME
    history -- display the last few commands

USAGE
    history [[-]ncommands]

PARAMETERS

    ncommands
        The number of commands to be displayed.

DESCRIPTION
    The HISTORY task prints a list of the last few commands executed. Only commands which terminated normally are included. The number of commands to be printed may be specified on the command line if desired. If the number is preceded by a minus sign the default number of lines is not changed, otherwise HISTORY will take the value given as the new default number of commands to be printed.

EXAMPLES
    1. Print the last few commands.

```
cl> history
```

    2. Print the entire history list.

```
cl> history -999
```

    3. Change the default number of history lines to be printed to 5 (and print the last five commands).

```
cl> history 5
```

    4. Save the history in the file "commands".

```
cl> history -999 > commands
```

SEE ALSO
    ehistory

NAME
    if else -- conditional execution of a statement

SYNTAX
    IF (expression) statement1 [ELSE statement2]

ELEMENTS

    expression
        A boolean valued expression.

    statement1, statement2
        The statements to be executed (possibly compound, i.e., enclosed in curly braces).

DESCRIPTION
    The IF statement is used to execute a statement only if the specified condition is true. An optional ELSE clause may be given to execute a different statement if the condition is false.

EXAMPLES
    1. Add X to Y only if X is less than Y.

```
        if (x < y)
            y += x
```

    2. If X is less than 10 print "small", else print "big".

```
        if (x < 10)
            print ("small")
        else
            print ("big")
```

    3. The ELSE IF construct.

```
        if (str == "+")
            val += x
        else if (str == "-")
            val -= x
        else if
            ...
```

    4. Nesting, use of braces.

```
        if (i > 0) {
            if (i < 10) {
                print ("0")
                sum = sum * 10 + i
```



```
                } else
                    print (" ")
            }
```

SEE ALSO
    for, case, break, next

NAME

intro -- a brief introduction to the CL

DESCRIPTION

1. GENERAL

The CL (or Command Language) is the command interpreter of the IRAF environment. Among it's responsibilities are: task initiation and termination; parameter retrieval and updating; and error handling. In addition the CL has certain 'builtin' utility functions which enable monitoring and changing of the IRAF environment, control flow features roughly modeled after C and Ratfor as well as fairly sophisticated capabilities for performing mathmatical calculations and string manipulation. The CL environment may easily be extended by the user.

2. TASK INITIATION AND TERMINATION

IRAF organizes tasks into groups called PACKAGES. When a package (which is itself a special kind of task) is invoked, it defines all the tasks which belong to that package, and the user may then execute any of the tasks in the package. Some of these new tasks may themselves be packages. Normally at the start of a CL session, the LANGUAGE package, including all functions built directly into the CL, and the SYSTEM package, which contains basic system utilities, are automatically invoked. The user may configure their "login.cl" file to automatically invoke other packages.

Within the CL a task is invoked by entering its name, e.g.

```
cl> reduce args
```

If two tasks in different packages have the same name, then the package name may be included:

```
cl> spectra.reduce args
```

The task name may be followed by a parameter list and tasks may be linked together by pipes (see parameters). The task initiates execution of either a script file, an ASCII file containing further CL commands, or an executable image, an external program linked with IRAF libraries so that it may be called as a sub-process from the CL process. The correspondence between the task name and the name of the script or image file is made using the task and redefine builtin commands.

When a script is run the CL effectively calls itself recursively with the new incarnation of the CL having its standard input not

from the terminal, but from the script file. When the script terminates the recursion unwinds, and the CL returns to an interactive mode. A script may itself call another script or executable.

An executable is run as a separate process with communication and synchronization maintained using an inter-process communication link (a pipe in UNIX or a shared memory region in VMS). When the executable requires a parameter a request is sent across the link, and the CL replies in the same fashion. When the process terminates it informs the CL and then hibernates. Normally the executable's process is not terminated, but is maintained in a process cache so that the executable may be used again without the overhead of reinitiating the process. The process finally terminates when the CL finishes, when the space in the cache is needed by a new executable image, or when the user explicitly clears the cache using the flprcache command. The size of the process cache is small, usually only three executables can be maintained in the cache.

3. PARAMETER RETRIEVAL AND UPDATE

Most CL tasks have a parameter list associated with them. When the task starts up, the CL looks to see if the user has a private copy of the parameters from the last time he ran this task. If so these parameters are loaded into memory. Otherwise the CL looks for the default values of the parameters and loads these. While the task is active the parameters are maintained in memory, but when it finishes the CL checks if any 'learned' parameters have been modified. If so a new private copy of the parameters is stored into the directory pointed to by the IRAF logical name 'uparm'. A number of builtins are used to control the handling of parameters including lparam, eparam, update and unlearn.

4. ERROR HANDLING

The CL attempts to trap most kinds of errors that may occur and to keep the user in a viable IRAF environment. When an error occurs in a script, execution of the script is terminated and the CL returns to an interactive level. The user may force an error using the ERROR builtin. When a executable image encounters an error it cannot handle itself, it sends an error message to the CL and then hibernates in the process cache until its next invocation. If executable was called by a script, the script is terminated and the CL returns to an interactive mode. The error message from the executable is relayed to the user.

SEE ALSO
    commands, mathfcns, strings

NAME
    jobs -- display the status of background jobs

USAGE
    jobs

PARAMETERS
    None.

DESCRIPTION
    JOBS is used to display the status of background jobs. If no job number is specified then all the status of all background jobs is displayed. For each job there is one line of output, e.g.

```
[2]  0:14 +Running  copy file1 file2 &
```

    Here 2 is the job number of the job; 0:14 is the clock time in minutes and seconds since the job was submitted; 'Running' indicates that the task is currently running while the '+' indicates that this was the last background job started. The remainder of the line is a copy of the actual command used to start the job.

    The possible states for a background job are:

```
Done    -- the job has finished normally
Exit N  -- the job terminated with exit code N
Stopped -- the job is waiting for input from the user
                (see the SERVICE command)
Running -- the job is currently executing
```

EXAMPLES

```
cl> jobs
    [1]  21:13  Done      mkhelp >& dev$null &
    [2]   0:05 +Running   count *.hlp > _junk &
```

BUGS
    Exit codes are rarely displayed when jobs terminate abnormally. The CL checks for background job termination only when a command is entered, hence the elapsed time shown will often be greater than it should be.

SEE ALSO
    kill, service

NAME
keep -- keep memory after task termination

USAGE
keep

DESCRIPTION
Normally when a script task terminates any tasks, packages, environment variables, etc. defined during the execution of that task are discarded (in other words, the memory used by the task is freed). The KEEP command instructs the CL to retain the definitions after script termination. Only one level of "keep" is achieved, e.g., if a script with a keep is called from a higher level script, then when the higher level script terminates the task definitions will still be lost (unless this higher level script also uses KEEP).

EXAMPLE
1. The most common use for KEEP is to retain a set of definitions in a script task.

```
set     pkdir = "home$hebrew/"
task    aleph, beth, kaph = hebrew.cl

keep
```

SEE ALSO
task, package

NAME
kill -- kill a background job

USAGE
kill job [job ...]

PARAMETERS

job A background job number, as returned by JOBS, or as printed when the job is submitted.

DESCRIPTION
KILL is used to forcibly terminate a background job. The user must specify the job number of the task to be killed. The job number is displayed when the job is started, and may also be seen using the JOBS command.

EXAMPLE
1. Kill job number 4.

```
cl> kill 4
```

SEE ALSO
jobs, service

NAME
    logging -- Using the CL logging features

DESCRIPTION
    The CL has some simple logging features to allow the recording of events of interactive sessions. From these saved event logs, one can trace a particular data analysis sequence, track errors in programs, and create new CL scripts. Other uses for the logfile exist as well.

    There are currently five types of logging messages, with a parameter to control what is actually logged. These include:

```
       commands - commands and keystrokes of an interactive session
     background - messages about and from background jobs
         errors - logging of error messages
          trace - start/stop trace of script and executable tasks
           user - user messages, via the PUTLOG builtin
```

    All of these types of messages except the interactive commands will show up as comments (i.e., starting with a '#') in the logfile. This facilitates using a previous logfile as input to the CL or as the basis for a script task.

    The CL parameters discussed below are used to control the logging features. These parameters can be set on the command line, in the "login.cl" file, or with the command "eparam cl".

PARAMETERS

    keeplog = no
        The overall on/off switch for the CL logging. When set to 'yes', the logfile will be opened and logging will commence. If the named logfile does not exist, it will be created, otherwise log messages will be appended to the existing file.

    logfile = "home$logfile"
        The name of the logfile.

    logmode = "commands nobackground noerrors notrace"

        LOGMODE controls what goes into the logfile. The following options are currently available:

        [no]commands
            Enables or disables logging of interactive commands. (This is usually always enabled.)

        [no]background
            Enables or disables background logging. This includes start/stop messages when background jobs are submitted and complete, as well as log messages from the background job itself.

        [no]errors
            Enables or disables error logging within script and executable tasks. If enabled, error messages printed on the terminal will also be logged.

        [no]trace
            Enables or disables tracing of script and executable tasks. If enabled, start and stop messages are logged, which include the package and task names, and the time. The start message also includes the filename of the task (.cl or .e).

EXAMPLES
    1. Turn all the logging features on except for background logging:

```
        cl> logmode = "commands nobackground errors trace"
```

BUGS
    Background logging to the same logfile can cause problems. The environment variable FILEWAIT should be set to 'no' to avoid file access conflicts. Even with this, reliability is not guaranteed and some messages will not get into the logfile.

SEE ALSO
    cl, putlog

NAME
    logout -- log out of the CL

USAGE
    logout

DESCRIPTION
    LOGOUT causes the CL to shut itself down, regardless of how many packages may currently be active. The only way to shut the CL down without killing it is to use LOGOUT; BYE is not allowed to shut the CL down, since it would be too easy to enter it by accident (and it takes a while to log back in).

    An error message will be printed if one attempts to log out of the CL while a device is still allocated. The device should be deallocated and the LOGOUT repeated, else type LOGOUT several times and you will be permitted to logout with the device still allocated.

EXAMPLE
    1. Logout of the CL.

```
        cl> logo
```

SEE ALSO
    deallocate, bye

NAME
    lparam -- list the parameters of a task or pset

USAGE
    lparam pset [pset ...]

PARAMETERS

    pset
        The name of the parameter set to be listed.

DESCRIPTION
    LPARAM lists one or more parameter sets. Psets are specified either by the name of the task with which the pset is associated, or by filename (pset files have the ".par" extension). If a file type pset is listed the extension must be included, since it is the presence or absence of the filename extension which LPARAM uses to distinguish between task-psets and named (file) psets.

    Each parameter is listed on a single line with the following format:

```
        param = value           prompt string
```

    Here "param" is the name of the parameter, "value" is the current value of the parameter (blank if undefined), and "prompt string" is the prompt for the parameter, if any. If the parameter is hidden, then the line is enclosed in parentheses. For arrays, instead of the values, a list of the dimensionalities is given. The EPARAM task may be used to examine or edit the contents of an array. When more than one task is listed the task name is prefixed to the list of each tasks parameters.

EXAMPLES
    1. List the parameter for the task DELETE. Note that the positional parameters are listed first, in the order in which they must be specified on the command line, followed by the hidden parameters.

```
cl> lparam delete
        files = "temp"          list of files to be deleted
     go_ahead = yes              ?
      (verify = no)             verify operation before deleting each file?
(default_acti = yes)            default delete action for verify query
 (allversions = yes)            delete all versions of each file
    (subfiles = yes)            delete any subfiles of each file
        (mode = "ql")
```

    2. List the contents of the file pset "delete.par". Named psets such as this are most commonly produced using the ":W FILENAME"

colon command in EPARAM, e.g., to prepare several different versions of the parameter set for a task.

```
cl> lparam delete.par
```

BUGS

You cannot list the parameters of a task that does not have a parameter file (e.g., all builtin tasks).

SEE ALSO

eparam, dparam, cache

NAME

mathfcns -- math functions available in the CL

SYNOPSIS

Function	Return value	Description
sin(x)	real	sine
cos(x)	real	cosine
tan(x)	real	tangent
atan2(x,y)	real	arc-tangent
exp(x)	real	e**x
log(x)	real	natural logarithm
log10(x)	real	common logarithm
frac(x)	real	fractional part
abs(x)	type of argument	absolute value
min(a,b,...)	type of min. arg	minimum of a list of values
max(a,b,...)	type of max. arg	maximum of a list of values
real(x)	real	convert to real
int(x)	integer	integer part

DESCRIPTION

A number of mathematical functions are available under the CL. In general they return real values and may be used wherever a real expression is valid. The input arguments may be integer or real and may be mixed in cases where the function has more than one argument. Exceptions:

abs(x)	returns real or integer depending on its argument
int(x)	returns an integer
min,max	return a copy of the min/max operand, no type change

Note that the intrinsic functions INT and REAL may be called to decode string valued arguments.

EXAMPLES

```
y = sin (x)
  = 180 / 3.1415927 * atan2 (x, y)
i = int (max (4.3, x, y, 2))
  = 1. - (sin(.5)**2 + cos(.5)**2)
```

BUGS

An invalid argument list to a math function (e.g. log(-1)) will terminate a script.

SEE ALSO
strings

NAME
mktemp -- make a unique file name

USAGE
mktemp root

PARAMETERS

root
The root (prefix) for the generated filename.

DESCRIPTION
MKTEMP returns a unique filename string which may be used to create a temporary file name. The string is the concatenation of three elements: the input argument, the process id, and a final character which changes on each call.

EXAMPLES
1. Create a unique filename with the root "sav" in the logical directory "tmp".

```
savefile = mktemp ("tmp$sav")
```

BUGS
Since some time may elapse between the creation of the filename and the creation of a file with that name, there is no guarantee that the name will still be unique when it is actually used, however the algorithm used to generate the name makes filename collisions unlikely.

NAME
next -- start the next iteration of a for or while loop

USAGE
next

DESCRIPTION
The NEXT statement begins the next iteration of the loop construct in which it is enclosed, whithout executing any of the statements remaining before the end of the loop.

EXAMPLES
1. Sum the pixels in a two dimensional array. Skip any negative valued pixels.

```
for (i=1;  i < NCOLS;  i+=1) {
    for (j=1;  j < NLINES;  j+=1) {
        if (pixel[i,j] < 0)
            next
        total += pixel[i,j]
    }
}
```

SEE ALSO
break, while, for

NAME
osfn -- convert an IRAF filename to a host system filename

USAGE
string = osfn (vfn)

PARAMETERS

vfn
The IRAF virtual filename to be translated into a host filename.

DESCRIPTION
OSFN is a string valued intrinsic function which takes an IRAF virtual filename as input and returns the equivalent host system filename as output. OSFN can only be called as a function.

EXAMPLES
1. Print the host equivalent of the vfn "hlib$login.cl".

```
cl> = osfn ("hlib$login.cl")
```

2. Compute a host filename for use as an argument to a foreign task (see help TASK for more information on foreign tasks).

```
cl> task $vdir = "$directory"   # VMS directory lister
cl> vdir /size osfn("bin$")
```

SEE ALSO
pathnames, task

NAME

package -- define a new package

USAGE

package pkgname

PARAMETERS

pkgname
: The name of the new package to be created. If called with no arguments, PACKAGE lists the currently defined packages in task search order.

DESCRIPTION

The PACKAGE task creates a new package. The newly defined package becomes the current package, and the prompt is changed to use the first two characters of the package name. The package command does not define any tasks within the package, that is done by subsequent TASK declarations. Subsequent TASK declarations will add tasks to the task list for the new package.

The new package remains the "current package" until another PACKAGE command is entered, or until the task in which the package command was entered is terminated. Normally PACKAGE will be used at the beginning of a script to define the package name. It will be followed by one or more task definitions, and then by a CL or CLBYE to interpret user commands, until the command BYE is entered by the user, at which time the package script task terminates, discarding the package and any associated definitions.

EXAMPLES

1. The use of PACKAGE in a package script task.

```
package lists

set     lists           = "pkg$lists/"

task    table,
        tokens,
        unique,
        lintran,
        columns,
        words           = "lists$x_lists.e"

task    $gcursor        = "lists$gcursor.cl"
task    $imcursor       = "lists$imcursor.cl"
task    average         = "lists$average.cl"
```



```
clbye()
```

2. List the currently defined packages in the order in which they will be searched for tasks.

```
cl> pack
clpackage
language
user
system
```

BUGS

All active packages must have unique names. To eliminate the possibility of parameter file name collisions in UPARM, the three chararacter string formed by concatenating the first two and final characters of the package name should be unique.

SEE ALSO

task, redefine

NAME

parameters -- IRAF parameters and their usage

DISCUSSION

1. INTRODUCTION

Parameters are the primary means of communicating information between the user and IRAF tasks, and between separate IRAF tasks. Each user effectively has their own copy of the parameters for the tasks they run, and by tailoring these as they wish, they may customize the IRAF environment. Here we describe characteristics of IRAF parameters. The syntax of parameter declarations is described elsewhere.

2. PARAMETER TYPES

The CL supports a variety of parameter datatypes, from the conventional string, integer, and floating point types, to the exotic struct and cursor types. There is no complex type in the CL.

char
: Character parameters are used to store strings of ASCII characters. By default character parameters have a maximum length of 64 characters, but this may be extended using the LENGTH option when the parameter is declared. A character parameter consisting of a single character can usually be treated as an integer, with a value equal to the ASCII value of the character.

int
: Integer parameters are used to store integer information. Integer parameters are stored internally as a long integer, permitting at least 32 bits of precision.

real
: Real parameters are stored internally as double's. In general they may be entered with or without a decimal point, and with or without an exponent. Note that the exponent should be entered using an E not a D.

bool
: Boolean parameters may only have the values YES or NO.

file
: File parameters are basically character parameters which are required to be valid file names. All operations legal on characters are legal on file parameters. Various checks on the accessibility or existence of a file may be automatically performed when a FILE type parameter is used at runtime.

struct
: Struct parameters are characters strings which are treated specially by the scan and fscan functions. Scan and fscan set structs to the remainder of the line being scanned without further parsing.

gcur, imcur
: The cursor parameters have a character string value with a predefined cursor value format. When a cursor type parameter is read in "query" mode, the hardware cursor on the graphics terminal or image display is physically read. If the cursor parameter is list-structured, cursor input may also be taken from a list (text file). For a more detailed discussion of cursor control in the CL, type HELP CURSORS.

3. LIST-DIRECTED PARAMETERS

Frequently one may have a list of values, e.g. numbers or file names, which one wishes to analyze in turn. To do this one may use a list-directed parameter. The parameter is defined with its value field set to the name of a file containing the list. The next time it is referenced its value will not be the string containing the file name, but rather the first value in the list. Subsequent calls will return later values in the list until an end-of-file is reached, at which point the parameter will appear to be undefined. The file may be rewound using the p_filename attribute of the parameter. Assigning the null string to a list parameter closes the associated list file.

```
int     *list = "listfile.lis"
int     cur_val

for (i=1;  i < nlist;  i+=1) {
    cur_val = list
    analyze (cur_val)
}
```

A common usage of struct list-directed parameters is to read files in conjunction with the FSCAN function. The following example prints out a file.

```
struct  *slist = "filer.lis"
struct  line

while (fscan (slist, line) != EOF)
    print (line)
```

4. MODES

The mode of a parameter determines two qualities: whether the parameter is prompted for when it is accessed, and whether the parameter is "learned", i.e. whether its value is saved between invocations of a task.

A hidden parameter is never prompted for unless it is undefined or has an illegal value. A query parameter is prompted for every time it is referenced, except that a query parameter which is set on a command line is not queried for when it is accessed within that task.

These are the two basic modes, but a parameter may also be defined to be automatic. This means that the parameter will use the mode not of the task, but of the package the task is part of, or by the CL. When an automatic parameter is referenced the CL searches up this hierarchy to find a mode which is not automatic and uses this for the mode. If the mode switch at all levels is automatic then the mode is set to hidden. The mode switch at the task, package and CL levels is determined by the VALUE, not the mode, of the parameter with the name "mode" associated with the task, package or CL.

Query and automatic parameters are learned by default, while hidden parameters are not.

5. RANGES

The CL supports ranges for integer and real variables, and enumeration lists for character strings. A user may specify either or both of a minimum and maximum for numbers, and the CL will reject any values which fall out of this range. Range checking is only performed during querying, or inside EPARAM, not when a value is assigned directly. For an enumerated string the input string is matched against any of the enumerated possibilities using a minimum-matching technique. A value with no match is rejected.

6. PARAMETER ATTRIBUTES

The user may access the different elements of a parameter using the parameter attributes. For some parameters certain of the attributes will be meaningless or undefined.

p_name
    The name of the parameter.

p_type
    A string indicating the basic type of the parameter:

```
        b       -- boolean
```



```
        i       -- int
        r       -- real
        s       -- string/char
        f       -- file
        struct  -- struct
        gcur    -- graphics cursor
        imcur   -- image cursor=
```

p_xtype
    This is the same as p_type except that the string is prefixed by "*" if the parameter is list directed.

p_mode
    A string indicating the mode of the parameter composed of the characters:

```
        q  --  query
        a  --  automatic
        h  --  hidden
        l  --  learned
```

p_value
    The value of the parameter. For a list-directed parameter this is a element in the file, not the file name. Generally this is what is accessed when the parameter attribute is not specified.

p_length
    For string type parameters (i.e. char, struct, file, gcur, imcur), the maximum length of the string.

p_mimimum
    The minimum value for a parameter. Also for enumerated strings the enumeration list.

p_maximum
    The maximum value for a parameter.

p_filename
    For list-directed parameters the file name associated with the parameter.

Attributes may appear on either side of an equals sign, e.g.

```
    list.p_filename = "test.fil"
    = str.p_length
    range = integ.p_maximum - integ.p_minimum
    list.p_xtype =
    = system.page.first_page.p_minimum      # Fully qualified.
```

It is illegal to assign to the p_name, p_type and p_xtype fields. Most of the direct use of the parameter attributes is expected to be in systems level programming.

7. ARRAYS

The user may define arrays of arbitrary dimensionality within the CL. The arrays are referenced in the conventional fashion with the index list enclosed in square brackets, and the individual elements separated by commas. In their internal representation, arrays are similar to those in Fortran, with the first element changing fastest as one traverses memory. The limits of each index may be specified.

In general the CL can only access one element of the array at a time but there is an automatic looping feature which permits the appearance of array arithmetic. Any executable statement in which an array is referenced but in which the exact element of the array is not defined (an "open" array reference) will cause the CL to implicitly execute that statement within a loop over all the elements of the array. More than one "open" array may appear in the expression but they agree on the limits of the loop. For example,

```
real x[20,20], y[20], z[10,20], t[20]

y = x[1,*]
t = log(y)
z = x[1:10,*]
```

8. SCOPE

A parameter is known via an implicit reference if the task in which it is defined is active. In an implicit reference the parameter name only, without a task or package qualifier, is given. The CL is always active, so that its parameters are always known. In a script, the script itself is active, so its parameters may be used implicitly. If the script calls another task, that sub-task may reference the invoking tasks parameters implicitly.

For an explicit reference, i.e. with task and package qualifiers, the parameter is known if the package in which the task is defined is active. For example, when starting the CL, the "lists" package is not active, thus the parameters of the "sort" task may not be referenced even in the form "lists.sort.param". However since the system package is activated during login to the CL, the parameters of "page" may be referenced by "page.param". In general a package qualifier is used only to remove ambiguity between tasks with the same name in two different packages.

9. STORAGE

There are several places in which parameters are stored. On disk the CL searches for the parameters for a task in three locations. For a procedure script, the default parameters are found in the script file itself, while other scripts and executables have a parameter file with defaults in the same directory as the script or executable. These default values are used the first time a task is run, or whenever the default values have been updated more recently than the user's copy of the parameters. The user's copy is created when a task terminates, and retains any "learned" changes to the parameters. It is created in a directory pointed to by the IRAF logical "uparm" which is usually a sub-directory of the default IRAF directory for the user.

The user may also use in-core storage for the parameters using the cache command. This keeps parameters for frequently used tasks available without requiring disk access. Cached parameters are copied to disk when the CL exits, or when the update command is used.

SEE ALSO

lparam, eparam, cache, unlearn, update, cursor

NAME
    prcache -- cache a subprocess

USAGE
    prcache task [task ...]

PARAMETERS

    task
        The name of a compiled IRAF task (not the filename of an executable file).

DESCRIPTION
    The CL maintains a small cache to store executable images. When the user invokes a task which calls an executable the cache is searched for the image before any attempt to load the executable image from disk. After completion of the task the CL retains the executable image in the process cache, until the space is needed by some other executable. Thus if a few commands are being executed frequently, the overhead of loading the image into core from disk is bypassed, which can result in a significant improvement in the response of the CL.

    By default, when the cache is full and a new executable must be run, the CL searches for the slot containing the task which has remained dormant the longest and replaces it with the new task.

    The PRCACHE command gives the user some control over this process. Using it without any arguments shows the tasks which are currently stored in the process cache. For each slot one gets a line like the following.

```
        [07] lyra!17763(4563X)           H  bin$x_images.e
```

    Here, 07 is the process slot number as required by FLPRCACHE to disconnect the process. The name "lyra" is the name of the node in the local network on which the process is executing; this is not normally the local node. In the example, 17763 (hex 4563X) is the process number (pid) of the executable. H indicates that the task is hibernating, i.e. the task was waiting in the cache to be invoked. R would show that the task was running. An L appended to either of these would show that the task had been locked into the cache by a previous prcache command. The last element on the line is the file name of the executable file which was loaded when the task was first invoked.

    If one or more task names are given as arguments, those tasks are locked into the cache, and will not be replaced by the CL without specific user intervention. If these tasks are not already in the cache, the corresponding executables are started, and the tasks are loaded into the cache.

    Note that the 'process cache' described here, and the 'parameter cache' described in the 'cache' command are entirely distinct, and a given task may be found in either, both, or neither of the two caches.

    Also note that only executable images reside in the process cache. Thus, for example, if the NEWS command is executed, it does not appear in the process cache, but the executable 'system$x_system.e' does, because NEWS calls PAGE, which is one of the many entries into the system executable.

    Locked process cache slots may only be freed with the FLPRCACHE command.

EXAMPLES
    1. Flush the system process and lock it back into the cache.

```
            cl> flpr dir
            cl> prc dir
```

    2. Print the current contents of the process cache.

```
            cl> prc
                [10] lyra!17764(4564X)           H  bin$x_plot.e
                [07] lyra!17763(4563X)           H  bin$x_images.e
                [04] lyra!17455(442FX)           HL bin$x_system.e
                       0
```

    3. Flush all processes which are not locked into the cache. This may be necessary after aborting a task to initialize (by re-executing) the associated process, which may not have recovered completely from the abort.

```
            cl> flpr
```

BUGS
    The user is responsible for making sure that he does not lock all the slots in the cache.

SEE ALSO
    flprcache

NAME
    fprint -- print to a parameter
    print -- print to the standard output

USAGE
    print expr [expr ...]
    fprint param expr [expr ...]

PARAMETERS

    expr
        Any expression, the string value of which is to be printed.

    param
        FPRINT will deposit the output string in the value field of this parameter.

DESCRIPTION
    The PRINT and FPRINT commands format a line of text and write it to either the standard output or in the case of FPRINT, the p_value field of the named parameter. The output is free format although spaces may be specifically inserted (as quoted string constants) to make the output easier to read. One space is automatically inserted after each numeric argument; this can be defeated by coercing the argument to a string with the STR intrinsic function. A newline is automatically output at the end of the output line. I/O redirection may be used with PRINT to write to a file.

    Compute mode (a parenthesized argument list) is recommended for this task to avoid surprises.

EXAMPLES
    1. Print the name of the current terminal.

```
cl> print ("terminal = ", envget ("terminal"))
```

    2. Output a blank line on the standard output, e.g., in a script.

```
print ("")
```

    3. Format a command and send it to the host system. In this example, "fname" is a string valued parameter.

```
cl> print ("!dir/full ", fname) | cl
```

    4. Write to a file.

```
for (x=1.;  x < 1E5;  x *= 10)
```



```
    print ("the sqrt of ", x, "is ", sqrt(x), >> "output")
```

BUGS
    The FPRINT task is not very useful since the same thing can be accomplished by string concatenation and assignment.

SEE ALSO
    scan, fscan

NAME

procedure -- declare a new CL procedure

SYNTAX

```
PROCEDURE proc_name [( [req_par, ...] )]

<query mode parameter declarations>
<hidden parameter declarations>

BEGIN
        <local variable declarations>
        <executable statements>
END
```

ELEMENTS

proc_name
: The name of the procedure. In the case of a procedure script, the script file should have the same name.

req_par
: A required (query mode) parameter for the procedure. Hidden parameters must be declared in the declarations section but do not appear in the argument list.

DESCRIPTION

The PROCEDURE statement is used to declare a new CL procedure. In the current CL, procedures are permitted only in ".cl" script files, and there may be only one procedure per file. The PROCEDURE statement must be the first non-comment statement in the script file. Any parameters which appear in the procedure argument list must be declared in the parameter declarations section as well and will default to mode "auto". Paremeters not in the required parameter list will default to mode "hidden". The order of positional parameters is the order in which the parameters appear in the argument list.

EXAMPLES

1. Declare a no-op procedure.

```
procedure noop
begin
end
```

2. A more complex procedure (hlib$devstatus.cl).

```
# DEVSTATUS -- Print status info for the named device.
```



```
procedure devstatus (device)

string  device  { prompt = "device for which status is desired" }
bool    verbose = no

string  logname, hostname
struct  *devlist
string  dev

begin
        dev = device
        _devstatus (dev)

        if (verbose) {
            # Print UNIX device status, too.

            devlist = "dev$devices"
            while (fscan (devlist, logname, hostname) != EOF) {
                if (logname == dev) {
                    print ("ls -l /dev/", hostname) | cl
                    break
                }
            }
            devlist = ""
        }
end
```

BUGS

CL procedures can only be placed in script files, they cannot currently be typed in interactively. Procedures cannot be precompiled. A procedure cannot return a function value. Arguments are passed only by value, not by reference. Procedure interpretation (and expression evaluation) is currently rather slow.

SEE ALSO

declarations, task

NAME
    putlog -- put a message to the logfile

USAGE
    putlog logmsg

PARAMETERS

    logmsg
        A message to append to the logfile.

DESCRIPTION
    PUTLOG is used to add user messages to the logfile. The CL parameter KEEPLOG must be set to 'yes' for this to take effect.

BUGS
    For executable tasks, the only way to call PUTLOG currently is via the low-level CLIO routine clcmd().

SEE ALSO
    cl, logging

NAME
    radix -- encode a number in any radix

USAGE
    string = radix (number, newradix)

PARAMETERS

    number
        The integer number to be encoded.

    newradix
        The radix or base in which the number is to be printed, e.g., 2 (binary), 8 (octal), 10 (decimal), 16 (hex), and so on.

DESCRIPTION
    RADIX is a string valued intrinsic function which formats an integer number in the indicated radix, return the encoded string as the function value. Note that the CL permits numbers to be input in octal or hex format (trailing B or X suffix respectively), allowing common numeric conversions to decimal to be done directly. The RADIX function is however the only CL function currently available for printing numbers in bases other than 10. RADIX can only be called as a function.

EXAMPLES
    1. Print the hex number 7cde in binary.

```
cl> = radix (7cdex, 2)
```

    2. Print the hex number 7cde in decimal.

```
cl> = 7cdex
```

    3. Print the number in variable I in decimal, octal, and hex.

```
cl> print (i, radix(i,8), " ", radix(i,16))
```

BUGS
    Very large bases produce strange results.

SEE ALSO
    print

NAME

task -- define a new IRAF task
redefine -- redefine an IRAF task

USAGE

```
task     t1 [t2 ...] = tfile
redefine t1 [t2 ...] = tfile
```

PARAMETERS

t1, t2, ...
: The names of the new logical tasks. The task name should be prefixed by a $ if the task has no parameter file. An optional extension should be appended if either the standard input or output of the task is a binary stream, rather than text. For example, "$mytask.tb" denotes a task with no parameter file, a text standard input, and a binary standard output.

tfile
: The name of the file to be executed or interpreted to run the task. The type of the task is determined by the file extension. An ".e" extension indicates an executable task, while ".cl" indicates a CL script task or procedure. The TFILE string is prefixed by a $ to define a FOREIGN TASK (see the discussion below).

DESCRIPTION

The TASK statement defines a new task to the CL, and is required before the task can be run from the CL. The new task is added to the "current package", i.e., the package that is listed when "?" is entered. Any task definitions made since the current package was entered will be discarded when the package is exited.

In addition to defining a new task, the TASK statement defines the type and attributes of the new task. Three types of tasks can be defined: script (.cl), executable (.e), and foreign ($...). A task is assumed to have a parameter file ("taskname.par", in the same directory as TFILE), unless the taskname is explicitly prefixed by a $. A suffix or extension may optionally be added to the task name to indicate whether the input and output streams are text or binary. The default is text, meaning that if output (or input) is redirected to a file, the file will be opened as a text file.

The FOREIGN TASK facility allows host system tasks, e.g., host utilities or user written Fortran or C programs, to be called from the CL as if they were regular IRAF tasks. The command line of a foreign task is parsed like that of any other task (and unlike an OS escape), allowing expression evaluation, i/o redirection, and background job submission. The difference between a regular IRAF

task and a foreign task is that the foreign tasks have little or no access to IRAF facilities, are usually machine dependent (and programs which use them are machine dependent), and cannot be cached. Nonetheless the foreign task facility is very useful for personalizing and extending the IRAF environment with a minimum of effort.

The TASK statement includes facilities for defining how the host system argument list for a foreign task will be built when the task is called from the CL. The simplest form of the foreign task statement is the following:

```
task [$]taskname = "$host_command_prefix"
```

where HOST_COMMAND_PREFIX is the first part of the command string to be passed to the host system. Any command line arguments are simply tacked onto the end of this string, delimited by blanks.

If this is insufficient then argument substitution may be used to define how the argument list is to be built up. The macro $N denotes argument N from the CL command line, with the first argument being number 1. The macro $0 is a special case, and is replaced the name of the task being executed. Likewise, $* denotes all arguments. If the character following the $ is enclosed in parenthesis, the corresponding argument string will be treated as an IRAF virtual filename, with the equivalent host system filename being substituted for use in the host command. Any other character sequences are passed on unchanged. The argument substitution macros are summarized in the table below.

$0	task name
$N	argument N
$*	all arguments
$(...)	host system filename translation of "..."

When a task is invoked, an executable is run by starting an attached sub-process, while a script is run by starting a new level of the CL with its standard input set to the script file.

An executable image may contain any number of executable CL tasks, hence it can be pointed to by multiple task names or in multiple TASK statements. A script file can only contain one script task.

REDEFINE has the same syntax as the TASK command, but all the task names must already be defined in the current package. It is often useful after misspelling the task file name in a task command.

EXAMPLES

1. Call up the editor to create a new program (task) mytask.x. Compile the new program. Declare it using the task statement and then run it.

```
cl> edit mytask.x                     # edit
cl> xc mytask.x                       # compile & link
cl> task $mytask = mytask.e           # define task
cl> mytask arg1 arg2                  # run it
```

2. Define a script task with associated parameter file (if the script is a PROCEDURE, the parameter file is omitted since procedure scripts always have defined parameters).

```
cl> task myscript = myscript.cl
```

3. Define the four new tasks implot, graph, showcap, and gkiextract. All have parameter files except showcap. The gkiextract task has a binary output stream. All tasks are executable and are stored in the executable file "plot$x_plot.e". Note the use of comma argument delimiters in this example; this is a compute mode example as would be found in a package script task.

```
task    implot,                 # compute mode syntax
        graph,
        $showcap,
        gkiextract.tb   = "plot$x_plot.e"
```

4. Make the listed UNIX programs available in the IRAF environment as foreign tasks. None of the tasks has a parameter file. The "$foreign" declares the tasks as foreign, and indicates that the IRAF task name is the same as the host system task name.

```
cl> task $ls $od $rlogin = $foreign
```

5. Define a couple of foreign tasks for VMS, where the command to be sent to VMS is not the same as the IRAF task name.

```
cl> task $run   = $run/nodebug
cl> task $debug = $run/debug
cl> task $top   = "$show proc/topcpu"
```

BUGS

The distinction between command and compute mode syntax can be confusing. When defining tasks in your login.cl or in a package script task, use compute mode, with commas between the arguments and all strings quoted (there are plenty of examples in the system). When typing in TASK statements interactively, use command mode. If you forget and leave in the commas, they will be assumed to be part of the task name, causing the following error message when the task is run:

```
ERROR: IRAF Main: command syntax error
```


SEE ALSO

prcache, flprcache, package

NAME
      set -- set the value of an IRAF environment variable
    reset -- reset (overwrite) the value of an IRAF environment variable

USAGE
    set [varname = valuestring]
    reset [varname = valuestring]

PARAMETERS

    varname
        The environment variable to be defined or set.

    valuestring
        The new string value of the environment variable.

DESCRIPTION

The CL maintains a list of environment variables, each of which consists of a keyword = value pair. The SET and RESET operators are used to define new environment variables, or to set new values for old environment variables. The two operators are equivalent with the exception that if the named environment variable is already defined, SET will push a new, temporary value for the variable, whereas RESET will overwrite the most recent definition of the variable. Environment variables may be examined using the SHOW task or the ENVGET intrinsic function.

A particular use for the environment variables is in the definition of IRAF logical names for directories. If an environment variable is set to a string corresponding to a system-dependent directory name, then the enviroment variable may then be used within the CL to refer to that directory.

For example,

```
set     testdir = "/usr/iraf/testdir"          # Unix
set     testdir = "dua2:[iraf.testdir]"        # VMS
task    tst1 = testdir$tst1.cl
```

New IRAF logicals may be defined in terms or existing IRAF logical names, i.e., logical names are recursively expanded.

```
set     subdir1 = testdir$subdir1/
task    tst2 = subdir1$tst2.e
```

If the SET command is entered without any arguments the current environment list is printed in the reverse of the order in which the definitions were made. If a variable has been redefined both the final and original definition are shown. The SHOW command can be

used to show only the current value.

EXAMPLES

1. Define the data directory "dd" on a remote node, and call IMPLOT to make plots of an image which resides in the remote directory.

```
cl> set dd = lyra!/u2/me/data
cl> implot dd$picture
```

2. Temporarily change the value of the variable PRINTER. The new value is discarded when the BYE is entered.

```
cl> cl
cl> set printer = qms
        ...
cl> bye
```

SEE ALSO
    show, envget

NAME

return -- exit a procedure

USAGE

return [value]

PARAMETERS

value
An optional value returned to the invoking procedure.

DESCRIPTION

The RETURN statement terminates a script and optionally returns a value to the invoking routine. Any number of RETURN statements may be present in a script. A RETURN statement without a value is equivalent to a BYE.

EXAMPLES

```
return
return (j+3)
```

BUGS

The return value cannot currently be utilized by the invoking procedure.

SEE ALSO

bye

NAME

scan -- read parameters from standard input
fscan -- read parameters from file, or another parameter
nscan -- get number of parameters scanned

USAGE

```
scan  (p1, p2, p3 ... pn)
fscan (param, p1, p2, p3, ... pn)

n = nscan()
```

PARAMETERS

pN The name of an output parameter, to receive a scanned value.

param
The name of the input parameter whose VALUE is to be scanned to produce the output values.

DESCRIPTION

SCAN and FSCAN permit the user to read in values from the standard input, a file, or a parameter and assign them to the listed parameters. FSCAN may also be used to read a string already in core. It is useful to consider these functions as performing two disjoint actions: acquiring a string, where SCAN and FSCAN differ; and parsing the string, where they are identical.

SCAN acquires its string by reading exactly one line from the standard input. The action of FSCAN depends on PARAM. If PARAM is a string, or a struct, then the string is simply the value of PARAM. If, however, PARAM is a list-directed struct, a call to FSCAN will get the next line from the file pointed to by PARAM. The file can be rewound by assigning a file name to PARAM. If either scan or fscan reach an EOF, they return with the value EOF and do not change any parameters.

Once the string has been acquired it is parsed into segments delimited by spaces or tabs. Scan and fscan do not recognize quoted strings, nor do they view ’,’ as a delimiter. Each token is then assigned in turn to p1 through pn. If there are too many tokens they are discarded, if there are too few, the corresponding parameters are not affected by the call. Any conversion error terminates the scan, but parameters already scanned retain their new values. An assignment to a struct terminates the scan because the entire unscanned portion of the string is assigned to the struct. Thus any struct should be the last parameter in a scan or fscan call.

Scan and fscan are intrinsic functions returning either EOF if end of file on the input list is sensed, or the number of parameters successfully scanned. The function NSCAN also returns the number of parameters successfully scanned in the last call to scan or fscan.

EXAMPLES

1. Print a list of radii, given a list of coordinates.

```
list = coords
while (fscan (list, x, y) != EOF)
    print (sqrt (x**2 + y**2))
```

BUGS

The syntax of scan and fscan is peculiar, in that they are the only functions where parameters are effectively passed by reference rather than by value. Thus p1, ... pn must be parameters whereas in similar contexts an arbitrary expression can be used wherever a parameter can.

SEE ALSO

string

NAME

service -- respond to a parameter request from a bkg job

USAGE

service [job]

PARAMETERS

job A background job number (defaults to 1).

DESCRIPTION

When a background job requires input from the terminal (e.g. if it queries for a parameter), the job enters a stopped state, and a message is displayed on the terminal. At the user's convenience, he should respond with a SERVICE command specifying the appropriate job number. The JOBS command can also be used to see what jobs require attention.

After entering the SERVICE command, any prompt sent by the background job is displayed, and the user may return a single line of input to the background task. Should more lines be needed several SERVICE calls may be necessary. The user may service jobs in any order, regardless of how the requests from the background jobs were received.

EXAMPLE

1. Respond to a parameter request from job 3.

```
cl> service 3
```

BUGS

If one never responds to a request for service from a background job, the job will eventually time out and abort. In principle it is possible to service queued background jobs as well as interactive (subprocess) background jobs, but in practice the request for service never reaches the terminal (and thus the user), hence all parameters should be specified before submitting a job to execute in a queue.

SEE ALSO

jobs, kill

NAME
    set -- set the value of an IRAF environment variable
    reset -- reset (overwrite) the value of an IRAF environment variable

USAGE
    set [varname = valuestring]
    reset [varname = valuestring]

PARAMETERS

varname
    The environment variable to be defined or set.

valuestring
    The new string value of the environment variable.

DESCRIPTION
The CL maintains a list of environment variables, each of which consists of a keyword = value pair. The SET and RESET operators are used to define new environment variables, or to set new values for old environment variables. The two operators are equivalent with the exception that if the named environment variable is already defined, SET will push a new, temporary value for the variable, whereas RESET will overwrite the most recent definition of the variable. Environment variables may be examined using the SHOW task or the ENVGET intrinsic function.

A particular use for the environment variables is in the definition of IRAF logical names for directories. If an environment variable is set to a string corresponding to a system-dependent directory name, then the enviroment variable may then be used within the CL to refer to that directory.

For example,

```
set     testdir = "/usr/iraf/testdir"           # Unix
set     testdir = "dua2:[iraf.testdir]"         # VMS
task    tst1 = testdir$tst1.cl
```

New IRAF logicals may be defined in terms or existing IRAF logical names, i.e., logical names are recursively expanded.

```
set     subdir1 = testdir$subdir1/
task    tst2 = subdir1$tst2.e
```

If the SET command is entered without any arguments the current environment list is printed in the reverse of the order in which the definitions were made. If a variable has been redefined both the final and original definition are shown. The SHOW command can be

used to show only the current value.

EXAMPLES
1. Define the data directory "dd" on a remote node, and call IMPLOT to make plots of an image which resides in the remote directory.

```
cl> set dd = lyra!/u2/me/data
cl> implot dd$picture
```

2. Temporarily change the value of the variable PRINTER. The new value is discarded when the BYE is entered.

```
cl> cl
cl> set printer = qms
        ...
cl> bye
```

SEE ALSO
    show, envget

NAME
    show -- show the current value of an IRAF environment variable

USAGE
    show [varname]

PARAMETERS

    varname
        The name of the environment variable to be displayed.

DESCRIPTION
    The SHOW command shows the current values of all defined environment variables if called with no arguments, or the value of a specific variable if an argument is given. Unlike SET, only current values are shown, not the entire history of the definitions of environment variables.

EXAMPLES
    1. Show the current default printer device.

```
cl> show printer
```

    2. Show all "std" (standard i/o stream) related variables.

```
cl> show | match std
```

SEE ALSO
    set

NAME
    sleep -- suspend process execution for the specified interval

USAGE
    sleep [nsec]

PARAMETERS

    nsec
        The number of seconds to sleep. Defaults to 0.

DESCRIPTION
    The SLEEP command causes the task to hibernate for the specified amount of time.

EXAMPLES
    1. Sleep for 10 seconds, and then ring the bell.

```
cl> sleep 10; beep
```

SEE ALSO
    wait

NAME
    strings -- string manipulation functions available in the CL

USAGE
```
str    (x)
substr (str, start, end)
stridx (test, str)
strlen (str)
```

DESCRIPTION
    The following functions are available for the manipulation of strings within the CL.

str (x)
    Converts its argument into a string. The argument may be boolean, integer or real.

substr (str, first, last)
    Extracts a substring from string STR. The first character in the string is at index 1.

stridx (test, str)
    Finds the position of the first occurence of any character found in TEST in the string STR, returning 0 if the match fails.

strlen (str)
    Returns the current length of a string. Note that the maximum length may be obtained as the value of the expression 'param.p_length'.

EXAMPLES
    1. Simple function calls.

```
s = str(y)                      # convert y to a string.
s = substr ("abcdefg", 2, 4)    # s = "bcd"
i = stridx ("abc", " eeboq")    # i = 4
i = strlen ("abc")              # i = 3
```

SEE ALSO
    scan, radix

NAME
    stty -- set/show terminal characteristics

USAGE
    stty [terminal]

PARAMETERS

terminal
    The logical name of the terminal to be used, i.e., the name of the device given in the DEV$TERMCAP file.

baud = 9600
    Set to some nonzero value to inform the VOS of the baud rate; the software does not automatically sense the baud rate. The baud rate must be known to accurately generate delays.

ncols = 80
    The logical width of the screen in characters; may be set to some value less than the physical width to produce more readable output on very high resolution terminals.

nlines = 24
    The logical height of the screen in characters.

show = no
    Show the current terminal driver settings. The SHOW function is automatically enabled if STTY is called with no arguments.

all = no
    Show all terminal driver settings, including those which are not currently in use. Setting ALL automatically sets SHOW.

reset = no
    Reset the terminal driver settings to their default (login time) values. Note that the terminal driver is not a task in the normal sense but is always active, and once a parameter is set the new value is retained indefinitely.

resize = no
    Recompute the terminal screen size parameters, TTYNCOLS and TTYNLINES, and update their values in the environment. If the terminal supports runtime querying of the screen size it will be queried (allowing the screen size to change dynamically at runtime), otherwise the values from the termcap entry for the terminal will be used.

clear = no
: Clear the function(s) which follow on the command line, e.g., "clear ucasein ucaseout" is equivalent to "ucasein=no ucaseout=no".

ucasein = no
: Map terminal input to lower case, e.g., when working on an old monocase terminal, or on a modern terminal with the shiftlock key on.

ucaseout = no
: Map terminal output to upper case.

login = "home$ttyin.log" [off]
: Log all input from the terminal to the named text file.

logio = "home$ttyio.log" [off]
: Log all terminal i/o to the named text file. May not be used if either LOGIN or LOGOUT mode is in effect, and vice versa.

logout = "home$ttyout.log" [off]
: Log all output to the terminal to the named text file.

playback = "home$ttyin.log" [off]
: Divert terminal driver input to the named "stty login" style text file, i.e., take input from a file instead of from the terminal. The effect is to exactly repeat a previous terminal session executed with LOGIN mode in effect, e.g., to test or demo software.

verify = no
: If VERIFY is enabled during PLAYBACK mode the terminal driver will read a key from the keyboard before executing each command in the logfile. Tap the space bar to execute the command, Q to terminate playback mode, or G to continue execution with VERIFY mode disabled. Typing any other key causes a help line to be printed.

delay = 500 (msec)
: If VERIFY is disabled during PLAYBACK mode the terminal driver will pause for DELAY milliseconds before executing each logfile command.

DESCRIPTION

The STTY task is used to set or display the terminal device characteristics and VOS terminal driver options. Without arguments, STTY prints the current characteristics of the terminal. The default terminal type can be changed by setting TTYNAME. The terminal characteristics NCOLS, NLINES or BAUD, may be changed by typing new values explicitly on the command line.

The most common use of STTY is to inform IRAF of the type of terminal being used, e.g.,

```
cl> stty vt100
```

would set the terminal type to "vt100". An error message will be printed unless an entry for the named terminal is present in the TERMCAP file; if the named terminal is a graphics terminal, there must also be an entry in the GRAPHCAP file.

To find out about the current terminal settings, type

```
cl> stty
```

or

```
cl> stty all
```

A limited number of terminal driver options may also be set. In particular, the VOS terminal driver (not to be confused with the host operating system terminal driver, a lower level facility) implements facilities for case conversion upon input or output, and for logging all i/o to the terminal and playing back a terminal session logged in a file.

CASE CONVERSIONS

The UCASEIN option, if set, will cause all upper case terminal input to be mapped to lower case (e.g., when working from an old monocase terminal). In this mode, individual upper case characters may be input by preceding them with the escape character ^, e.g., "^MAKEFILE" equates to "Makefile". The full set of ^ escapes is summarized below. The option UCASEOUT will cause all terminal output to be mapped to upper case. Preceding either or both of these option keywords by CLEAR causes the options to be cleared.

```
^       shift next character to upper case
^+      shift lock (caps lock)
^-      clear shift lock
^^      the character ^
```

Case shifting is disabled in raw mode, e.g., while in cursor mode, and in EPARAM. All standard IRAF software, however, will sense that ucase mode is set before entering raw mode, and will behave as expected. Ucase mode is also disabled by the STDGRAPH kernel whenever the graphics workstation is activated.

Note that ^ is also the history metacharacter, hence ^^ must be used when in UCASEIN mode to pass a single ^ to the CL history mechanism. In cursor mode, upper case keystrokes are intercepted by cursor mode unless escaped with a backslash. Escaped keystrokes are mapped to lower case by cursor mode if UCASEIN mode is in effect, terminating cursor mode and returning a lowercase key to the applications program.

RECORDING TERMINAL I/O

The terminal driver options LOGIO, LOGOUT, and LOGIN may be used to log, respectively, all terminal i/o, all output to the terminal, or all input from the terminal. The logfile names are "home$ttyin.log", "home$ttyout.log", or "home$ttyio.log", unless a different logfile name is specified by the user. All logfiles are standard textfiles containing only printable characters.

Terminal i/o logging is especially useful for debugging TERMCAP and GRAPHCAP entries for new terminals. All IRAF terminal i/o is logged, including raw mode i/o and graphics output. Terminal i/o from foreign tasks or OS escapes is not logged since these tasks bypass the VOS to talk directly to the user terminal.

Each sequence of characters read from or written to the terminal (via a zgettt or zputtt call to the driver) appears as one logical line of text in the logfile, delimited by the data character \n (newline). When reading from a terminal in raw mode, each input character will appear on a separate line in the logfile with no newline, since only a single data character is read at a time during raw mode input. All control characters embedded in the data, including newline terminators, are rendered into printable form. Long lines are broken near the right margin, adding an escaped newline and indenting continuation lines 4 spaces.

Terminal i/o logging is intended primarily for debugging purposes, rather than for logging user commands; the IRAF command language provides a more user friendly facility for command logging (see the LANGUAGE.LOGGING manpage for further information on the CL command logging facilities). All unprintable ASCII codes are rendered in the logfile in a printable form intended to eliminate any ambiguity regarding the exact sequence of characters sent to or received from the terminal. In addition to the standard escape sequences \n, \t, \r, etc., the following special escape sequences are used:

\\	\
\^	^
^@	NUL (ascii 000)
^[A-Z]	ctrl/a - ctrl/z (ascii 001 - 032)
^[	ESC (ascii 033)
^\	FS (ascii 034)
^]	GS (ascii 035)
^^	RS (ascii 036)
^_	US (ascii 037)
\s	blank (ascii 040)
\<newline>	long i/o record continued on next line

These special escape sequences, plus any ordinary characters, constitute the DATA recorded in the logfile. The following additional escape sequences are used to record information about the logging session itself in the logfile.

\#	rest of line is a comment
\T	terminal device name at log time
\G	stdgraph device name at log time
\O	timestamp written at start of log session

Any whitespace (unescaped blanks, tabs, or newlines) appearing in the logfile is put there only to make the file more readable, and is not considered data. Blocks of text may be enclosed in a logfile delimited by escaped curly brackets, i.e., "\{ ... \}". This is used for the PLAYBACK facility described in the next section.

PLAYBACK OF TERMINAL SESSIONS

The terminal driver has the capability not only of recording terminal i/o in a file, but of taking input from a logfile to repeat a sequence of commands previously entered by the user with terminal input logging enabled. Note that we are not talking about simply playing back recorded output, but of actually executing an arbitrary sequence of commands formerly entered by the user. This is different from executing a sequence of commands entered into, for example, a CL script, because ALL input is recorded, including not only the commands, but also all responses to parameter queries, all rawmode keystroke input, and all graphics cursor input occuring interactively during execution of the recorded commands. These PLAYBACK SCRIPTS are useful for preparing automated demos or tutorials of complex software, and for preparing scripts to be used to automatically test software.

The basic sequence used to record and later playback a terminal session is as follows:

```
cl> stty login [= logfilename]
        <execute an arbitrary sequence of commands>
cl> stty clear login                    # or stty reset
cl> stty playback [= logfilename]
```

Naturally, the playback script must be executed in the same context as when the script was generated, i.e., one must ensure that all necessary packages have been loaded, that the current directory has been set to the proper value if it matters, and so on. It is not necessary to execute a playback script on the same type of video terminal or graphics terminal as was used when the script was recorded; since only the terminal input is being recorded, playback scripts are device independent and may be played back on any terminal.

If desired the commands necessary to establish the starting context may be recorded as part of the script. If the script is going to be repeatedly executed it may also be desirable to include commands at the end of the recording session to clean up, e.g., deleting any temporary files created during the recording session. If anything has changed which causes a command to abort during execution of a playback script, normal terminal input is automatically restored, aborting the script. Note that if the "stty playback" command gets into the playback script for some reason, e.g., because the "stty reset" (or "stty login=no" etc.) was omitted, then the script will repeat indefinitely. This may or may not be what was desired.

Two STTY command line arguments are provided for controlling the execution of a playback script. By default, when a script is played back the terminal driver will pause for DELAY milliseconds after echoing the command to be executed, to give the user watching the playback a chance to read the command. Aside from this programmed delay, execution is fully automated. For example,

```
cl> stty play=filename delay=2000
```

would playback the file "filename", with a delay of 2 seconds following echo of each line of recorded input text.

Alternatively, the user may request that the driver pause and wait for the user to type a key before executing each logged command (i.e., before returning each input line of text to the application). This is called the VERIFY option. In verify mode, the following keystrokes may be entered to continue execution:

space, return	continue execution
’g’	go: turn verify mode off and continue
’q’	quit: terminate playback mode

Verify mode is automatically disabled during raw mode input since the input consists of single characters and an inordinate number of verification keystrokes would be required from the user. Either of the VERIFY or DELAY options may be overridden by control directives embedded in the playback text, as we shall see in the next section.

CUSTOMIZING PLAYBACK SCRIPTS

Although playback scripts may be and often are generated and played back without ever looking at or modifying the actual playback script, there are cases where it may be desirable to do so. For example, when generating a script to be used as a demo or tutorial, it may be desirable to insert explanatory text into the script to be printed out on the terminal when the script is played back, to explain to the person running the script what is going on. Likewise, it may be desirable to control the verify and delay options at a granularity finer than the entire script.

Explanatory text and/or playback control directives may be inserted into the script using the following construct:

```
"\{" [<control_directives>] [<text>] "\}"
```


where CONTROL_DIRECTIVE refers to one of the following:

%V+	turn verify on
%V-	turn verify off
%NNN	set DELAY to NNN milliseconds

For example,

```
dir\{%5000
[list the current directory]\}\n
```

would cause the following to be output, followed after a 5 second delay by a listing of the current directory (the "<>" is not printed, but shows where the cursor will be during the 5 second pause):

```
cl> dir
[list the current directory]<>
```

Note that the newline following the "\{%5000" in the above example is textual data, and will be output to the terminal along with whatever follows, up until the closing brace, i.e., "\}". The amount of text to be output may be arbitrarily large; there is a builtin limit (currently 4096 characters), but it is unlikely that this limit will ever be exceeded, since no more than one pageful of text should ever be output in a single call.

Normally, a %V or %NNN control directive refers only to the input record with which the enclosing \{...\} control block is associated. The global value of VERIFY or DELAY is temporarily overridden for the current record. If desired, the global value may instead be permanently modified by adding a ! after the %, e.g.,

```
\{%!V-%3000...\}
```

would permanently disable VERIFY (unless a %V+ or %!V+ directive follows later in the script) then output the text "..." followed by a 3 second delay.

To know where to insert the control directives into a script, it is important to understand that input from the script is RECORD ORIENTED, and that a control directive refers to the input record with which it is associated. An input record is a single LOGICAL line of text in the input file. Note that a logical line of text may span multiple physical lines, if the newlines are escaped or present as textual data within a control directive. The position of the control directive within the input record determines where the explanatory text will be positioned relative to the input text, when both are echoed to the terminal. Any programmed delay or pause will always occur after echoing the full record on the terminal.

RAW MODE PLAYBACK

When a program is executing which reads from the terminal in raw mode, each character is read from the terminal as soon as it is typed, and input characters are not echoed to the terminal unless the application explicly does the echoing. Examples of programs which use raw mode input are EPARAM and PAGE, which are keystroke driven, and any program which reads the GRAPHICS CURSOR, since a graphics cursor read uses raw mode input.

Playback works much the same for raw input mode as for line input mode, except that during raw mode input the input records normally consist of single characters, rather than entire lines of text. By default, VERIFY is turned off while reading from the terminal in raw mode, to avoid having the user verify each individual character. Also, the terminal driver will not echo text read from the playback file in raw mode, since the text would not have been echoed if playback were not in effect.

CURSOR READS IN PLAYBACK MODE

A typical Tektronix style cursor read will look something like the following, as recorded in an STTY LOGIN script file following a recording session:

```
K
3
)
'
*
\r
```

This six character sequence consists of the key value of the cursor read (K), followed by the [x,y] cursor coordinate encoded as four ascii characters ("3)'*" in this case), followed by the "GIN mode terminator" character or characters, normally a single CR (\r). Of course, if the terminal is not a Tektronix compatible terminal (e.g., DEC-Regis), the details will differ from this example.

The single character per line format of a cursor read reflects the fact that each input record is a single character when reading from the terminal in raw mode. For the purposes of playback, however, such a sequence may be reformatted on a single line if desired, to improve the readibility of a script (the extra whitespace in the second example is ignored, since if a space were data it would appear as \s).


```
        K3)'*\r
or
        K 3 ) ' * \r
or
        K
        3)'*
        \r
etc.
```

To set the values of the VERIFY or DELAY parameters for a cursor read one may insert the \{...\} sequence anywhere before the \r delimiter is returned to the application, e.g.,

```
K3)'*\r\{%V+\}
```

would do, since the sequence shown forms one logical input record in the playback file, and the control directive included will be processed before any input data characters from the record are returned to the application. If the multi-line form of a cursor read is used, the control directive may be tacked onto any of the records K through \r in the example.

Output of explanatory text in an interactive graphics session is a little more tricky, since if one is not careful the text will come out while in graphics mode, causing it to be rendered as random lines drawn all over the screen. A safe technique for outputting comments during playback of a graphics session is to output the text to the STATUS LINE, taking care of course to output only a single line of text at once (since multiple lines written to the status line would rapidly flash by, leaving only the last line visible on the screen). We can do this by taking advantage of the : command sequence, which can be used to put the terminal temporarily into status line output mode.

```
:####\r
\{%5000
This is a status line comment\}
^U\177
```

For example, insertion of the above sequence between any two cursor reads in a recorded interactive graphics session would cause the text "This is a status line comment" to be written to the status line, with normal execution of the script occurring after a 5 second delay followed by erasure of the status line and exit from status line mode (due to the ctrl/u and rubout inserted as data after the colon cursor read).

While executing an interactive graphics session via playback, cursor values are read from the playback script instead of from the terminal, hence the user never sees the actual cursor crosshairs on the screen. To give the user some idea of what is going on, the key values of successive cursor mode keystrokes are echoed in ascii down the left side of the screen, starting at the upper left. The keystroke value is also echoed at the position of the cursor, to indicate where the cursor crosshairs would have been in an actual interactive session.

SAMPLE PLAYBACK SCRIPT

We conclude with an example of a complete playback script which can be entered into a file and played back to demonstrate some of the features of the IMPLOT task in the PLOT package (the PLOT package must already be loaded).

```
\O=NOAO/IRAF V2.6 iraf@pavo Fri 20:09:21 01-Jan-88
\T=gterm40
\G=gterm
\n
imheader\sdev$pix\slo+\suser-\n\{%3000
[Print image header]\}
\n
implot\sdev$pix\n
J3..8\r J3-,)\r J3+)9\r K3)’*\r J3((0\r l3&’;\r
:####\r
\{%5000
[use key ‘o’ to overplot]\}
^U\177
o3&’;\r
K3&’;\r K3%*(\r K3#,3\r l3!.?\r
:####\r
\{%5000
[key ‘X’ expands the plot in x]\}
^U\177
X3!.?\r
qXXXX\r
stty\sreset\n
```

EXAMPLES

1. Show the current terminal type and attributes.

```
cl> stty
Terminal=vt640, ncols=80, nlines=24, 9600 baud
ucasein=no, ucaseout=no, logio=off
```

2. Tell the system that the terminal is a vt100.

```
cl> stty vt100
```

3. Set the baud rate of the current terminal to 9600 baud.

```
cl> stty baud=9600
```

4. Set the width of the screen to 80 columns, e.g., to get short

menus on a workstation where the physical number of columns may be much greater than 80.

```
cl> stty ncols=80
```

5. Set the terminal type to 4012 and set ucasein and ucaseout modes.

```
cl> stty 4012 ucasein ucaseout
```

6. Clear the ucasein and ucaseout modes.

```
cl> stty clear ucasein ucaseout
```

7. Record a terminal session in the default logfile (home$ttyio.log).

```
cl> stty logio
```

8. Record input from the terminal in the file "demo".

```
cl> stty login=demo
```

9. Terminate logging and playback the terminal session recorded in this file.

```
cl> stty reset
cl> stty playback=demo
```

BUGS

1. Note that, when working with a terminal which supports runtime querying of the screen size, the screen size is queried when the STTY RESIZE command is executed, rather than when the terminal screen actually changes size. Hence, the screen size parameters printed by a command such as STTY SHOW will not necessarily reflect the actual screen size. STTY RESIZE SHOW queries the terminal for the screen size, hence should always be correct. The screen size is automatically queried whenever the PAGE or HELP tasks are run.

2. The terminal screen size is determined by querying the terminal for the screen size, and reading the response back (this technique has the advantage that it works remotely over IPC and network connections, and is host system independent). If the terminal does not respond for some reason, e.g., because the terminal type has been set improperly and the terminal does not support the query function, then STTY will hang. Typing a carriage return causes execution to resume, after which the error should be corrected.

SEE ALSO

language.logging, fio$zfiott.x, etc$sttyco.x

NAME

    switch -- switch case statement

SYNTAX

```
switch (expr) {
case val1 [, val1,...]:
    statements
case val3 [, val3,...]:
    statements
        (etc.)
default:
    statements
}
```

ELEMENTS

expr

An integer-valued expression tested before entry into the switch block.

valN

Integer valued constants used to match expression.

statements

Simple or compound statements to be executed when the appropriate case or default block is selected.

DESCRIPTION

The SWITCH statement provides a multiway branch capability. The switch expression is evaluated and control branches to the matching CASE block. If there is no match the DEFAULT block, if present, receives control. If no DEFAULT block is present, the switch is skipped.

Each CASE statement consists of a list of values defining the case, and an executable statement (possibly compound) to be executed if the case is selected by the switch. Execution will continue until the next case is reached, at which time a branch out of the SWITCH statement occurs. Note this difference from the C switch case, where an explicit BREAK statement is required to exit a switch. If a BREAK is used in a CL switch, it will act upon the loop statement containing the switch, not the switch itself.

Note that both the switch expression and the case constants may be integers, or single characters which are evaluated to their ASCII equivalents.

The DEFAULT statement must be the last statement in the switch block.

SWITCH (Feb86)   language   SWITCH (Feb86)

EXAMPLES

1. Multiple cases, no default case.

```
switch (opcode) {
case 1:
    task1 (args)
case 2:
    task2 (args)
case 5:
    task5 (args)
}
```

2. Multiple values in a case.

```
switch (digit) {
case '1','2','3','4','5','6','7':
    n = n * 8 + digit - '0'
default:
    error (1, "invalid number")
}
```

BUGS

Only integer values are allowed (no strings). The case values must be constants; ranges are not permitted.

SEE ALSO

if else, goto

NAME

task   -- define a new IRAF task
redefine -- redefine an IRAF task

USAGE

```
task     t1 [t2 ...] = tfile
redefine t1 [t2 ...] = tfile
```

PARAMETERS

t1, t2, ...
: The names of the new logical tasks. The task name should be prefixed by a $ if the task has no parameter file. An optional extension should be appended if either the standard input or output of the task is a binary stream, rather than text. For example, "$mytask.tb" denotes a task with no parameter file, a text standard input, and a binary standard output.

tfile
: The name of the file to be executed or interpreted to run the task. The type of the task is determined by the file extension. An ".e" extension indicates an executable task, while ".cl" indicates a CL script task or procedure. The TFILE string is prefixed by a $ to define a FOREIGN TASK (see the discussion below).

DESCRIPTION

The TASK statement defines a new task to the CL, and is required before the task can be run from the CL. The new task is added to the "current package", i.e., the package that is listed when "?" is entered. Any task definitions made since the current package was entered will be discarded when the package is exited.

In addition to defining a new task, the TASK statement defines the type and attributes of the new task. Three types of tasks can be defined: script (.cl), executable (.e), and foreign ($...). A task is assumed to have a parameter file ("taskname.par", in the same directory as TFILE), unless the taskname is explicitly prefixed by a $. A suffix or extension may optionally be added to the task name to indicate whether the input and output streams are text or binary. The default is text, meaning that if output (or input) is redirected to a file, the file will be opened as a text file.

The FOREIGN TASK facility allows host system tasks, e.g., host utilities or user written Fortran or C programs, to be called from the CL as if they were regular IRAF tasks. The command line of a foreign task is parsed like that of any other task (and unlike an OS escape), allowing expression evaluation, i/o redirection, and background job submission. The difference between a regular IRAF

TASK (Apr87)   language   TASK (Apr87)

task and a foreign task is that the foreign tasks have little or no access to IRAF facilities, are usually machine dependent (and programs which use them are machine dependent), and cannot be cached. Nonetheless the foreign task facility is very useful for personalizing and extending the IRAF environment with a minimum of effort.

The TASK statement includes facilities for defining how the host system argument list for a foreign task will be built when the task is called from the CL. The simplest form of the foreign task statement is the following:

```
task [$]taskname = "$host_command_prefix"
```

where HOST_COMMAND_PREFIX is the first part of the command string to be passed to the host system. Any command line arguments are simply tacked onto the end of this string, delimited by blanks.

If this is insufficient then argument substitution may be used to define how the argument list is to be built up. The macro $N denotes argument N from the CL command line, with the first argument being number 1. The macro $0 is a special case, and is replaced the name of the task being executed. Likewise, $* denotes all arguments. If the character following the $ is enclosed in parenthesis, the corresponding argument string will be treated as an IRAF virtual filename, with the equivalent host system filename being substituted for use in the host command. Any other character sequences are passed on unchanged. The argument substitution macros are summarized in the table below.

$0	task name
$N	argument N
$*	all arguments
$(...)	host system filename translation of "..."

When a task is invoked, an executable is run by starting an attached sub-process, while a script is run by starting a new level of the CL with its standard input set to the script file.

An executable image may contain any number of executable CL tasks, hence it can be pointed to by multiple task names or in multiple TASK statements. A script file can only contain one script task.

REDEFINE has the same syntax as the TASK command, but all the task names must already be defined in the current package. It is often useful after misspelling the task file name in a task command.

EXAMPLES

1. Call up the editor to create a new program (task) mytask.x. Compile the new program. Declare it using the task statement and then run it.

```
cl> edit mytask.x                       # edit
cl> xc mytask.x                         # compile & link
cl> task $mytask = mytask.e             # define task
cl> mytask arg1 arg2                    # run it
```

2. Define a script task with associated parameter file (if the script is a PROCEDURE, the parameter file is omitted since procedure scripts always have defined parameters).

```
cl> task myscript = myscript.cl
```

3. Define the four new tasks implot, graph, showcap, and gkiextract. All have parameter files except showcap. The gkiextract task has a binary output stream. All tasks are executable and are stored in the executable file "plot$x_plot.e". Note the use of comma argument delimiters in this example; this is a compute mode example as would be found in a package script task.

```
task    implot,                 # compute mode syntax
        graph,
        $showcap,
        gkiextract.tb   = "plot$x_plot.e"
```

4. Make the listed UNIX programs available in the IRAF environment as foreign tasks. None of the tasks has a parameter file. The "$foreign" declares the tasks as foreign, and indicates that the IRAF task name is the same as the host system task name.

```
cl> task $ls $od $rlogin = $foreign
```

5. Define a couple of foreign tasks for VMS, where the command to be sent to VMS is not the same as the IRAF task name.

```
cl> task $run   = $run/nodebug
cl> task $debug = $run/debug
cl> task $top   = "$show proc/topcpu"
```

BUGS

The distinction between command and compute mode syntax can be confusing. When defining tasks in your login.cl or in a package script task, use compute mode, with commas between the arguments and all strings quoted (there are plenty of examples in the system). When typing in TASK statements interactively, use command mode. If you forget and leave in the commas, they will be assumed to be part of the task name, causing the following error message when the task is run:

```
ERROR: IRAF Main: command syntax error
```


SEE ALSO

prcache, flprcache, package

NAME
    time -- display the current time

USAGE
    time

DESCRIPTION
    TIME writes the current time and date on the standard output.

EXAMPLE

```
cl> time
Fri 12:50:29 14-Feb-86
```

SEE ALSO
    sleep, wait, jobs

NAME
    unlearn -- restore initial defaults for parameters

USAGE
    unlearn name [name ...]

PARAMETERS

name
    An IRAF task or package name.

DESCRIPTION

Normally when a task terminates the values of the query mode task parameters used are stored in the parameter file on disk, appearing as the new defaults the next time the task is run. The UNLEARN command instructs the CL to forget any task parameters it might have learned and to use the initial default values the next time the task is run. If a tasks parameters have been cached, then they are removed from the parameter cache.

If a package name is specified all the tasks in the package are unlearned.

EXAMPLES

1. Unlearn the parameters for the delete and plot.graph tasks.

```
cl> unlearn delete plot.graph
```

2. Unlearn the parameters for all tasks in the DATAIO package.

```
cl> unlearn dataio
```

3. To unlearn the parameters for all tasks in the system, log out of the CL and run MKIRAF, or enter the following:

```
cl> chdir uparm
cl> delete *.par
```

BUGS

It is possible for the parameter set for a task to become corrupted, e.g., if the CL is interrupted while it is updating the parameter file on disk, causing a truncated file to be written. If this should occur one will get error messages complaining about illegal arguments or parameters not found when the task is run. The fix is to "unlearn" the parameters for the task.

When the CL fetches the parameters for a task, it checks to see if the system defaults have been updated more recently than the user's copy of the parameter set, and uses the system copy if it is more recent, after printing a message to warn the user. This is done by comparing the file dates for the system and user parameter sets. On VMS, it is easy for the modify date of the system copy of the parameter set to become updated even though the file data has not been modified, causing an annoying warning message to be printed when the task is later run. Should this occur, the best solution is to unlearn all affected parameter sets.

SEE ALSO

cache, update, lparam, eparam

NAME

update -- update the parameters for a task on disk

USAGE

update task [task ...]

PARAMETERS

task
    An IRAF task name.

DESCRIPTION

Normally when a task terminates the values of the task parameters used are stored for the next invocation of the task in a disk file in the users UPARM directory. However if the task parameters have been cached by the CACHE command, this will not be done until the CL terminates. In the case of a background job, automatic updating of parameters is disabled. The UPDATE command is used to force the parameters for a task to be updated on disk.

EXAMPLE

1. Update the parameters for the PAGE task.

```
cl> update page
```

BUGS

The parameter set is only updated on disk if a parameter has been modified since the last update.

SEE ALSO

cache, unlearn

NAME
wait -- wait for a background job to terminate

USAGE
wait [job job ...]

PARAMETERS

job A background job number, as printed when the job is submitted, or as given by the JOBS command.

DESCRIPTION
The WAIT task causes the CL to hibernate until a background job or jobs terminates. No arguments, or a job number of 0 means to wait until all background jobs finish, while other arguments can be specified to wait for a particular job. If a background job is not running the wait returns immediately.

EXAMPLES
1. Wait for any background jobs to finish, beeping the terminal when done.

```
cl> wait;beep
```

2. Wait for job 3 to terminate.

```
cl> wait 3
```

BUGS
Deadlock is possible.

SEE ALSO
jobs, kill, service

NAME
while -- while loop construct

SYNTAX
while (expression) statement

ELEMENTS

while
: Required keyword.

expression
: A boolean valued expression tested before each iteration.

statement
: A statement (possibly compound) to be executed in each iteration of the loop.

DESCRIPTION
The WHILE loop executes the enclosed statements while the specified condition is true.

EXAMPLES
1. An infinite loop.

```
while (yes) {
    sleep 30
    time
}
```

2. Type a file.

```
list = "home$login.cl"
while (fscan (list, line) != EOF)
    print (line)
```

SEE ALSO
for, case, break, next